

Alpha Architecture Reference Manual

Fourth Edition

This document is directly derived from the internal-only Alpha System Reference Manual and is an accurate and complete description of the Alpha architecture. It is available at the following anonymous FTP site: [ftp.compaq.com/pub/products/alphaCPUdocs/alpha_arch_ref.pdf](ftp://ftp.compaq.com/pub/products/alphaCPUdocs/alpha_arch_ref.pdf)

COMPAQ

Compaq Computer Corporation

January 2002

The information in this publication is subject to change without notice.

COMPAQ COMPUTER CORPORATION SHALL NOT BE LIABLE FOR TECHNICAL OR EDITORIAL ERRORS OR OMISSIONS CONTAINED HEREIN, NOR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL. THIS INFORMATION IS PROVIDED "AS IS" AND COMPAQ COMPUTER CORPORATION DISCLAIMS ANY WARRANTIES, EXPRESS, IMPLIED OR STATUTORY AND EXPRESSLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR PARTICULAR PURPOSE, GOOD TITLE AND AGAINST INFRINGEMENT.

This publication contains information protected by copyright. No part of this publication may be photocopied or reproduced in any form without prior written consent from Compaq Computer Corporation.

© Compaq Computer Corporation 2002.
All rights reserved. Printed in the U.S.A.

The software described in this publication is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement.

COMPAQ, the Compaq logo, Himalaya, NonStop, and VAX Registered in United States Patent and Trademark Office.

Alpha and OpenVMS are trademarks of Compaq Information Technologies Group, L.P in the United States and other countries.

UNIX is a trademark of The Open Group.

Other product names mentioned herein may be trademarks and/or registered trademarks of their respective companies.

Contents

Preface

1 Introduction to the Common Architecture (I)

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

2 Basic Architecture (I)

2.1	Addressing	2-1 (I)
2.2	Data Types	2-1 (I)
2.2.1	Byte	2-1 (I)
2.2.2	Word	2-1 (I)
2.2.3	Longword	2-2 (I)
2.2.4	Quadword	2-2 (I)
2.2.5	VAX Floating-Point Formats	2-3 (I)
2.2.5.1	F_floating	2-3 (I)
2.2.5.2	G_floating	2-4 (I)
2.2.5.3	D_floating	2-5 (I)
2.2.6	IEEE Floating-Point Formats	2-6 (I)
2.2.6.1	S_floating	2-6 (I)
2.2.6.2	T_floating	2-8 (I)
2.2.6.3	X_floating	2-9 (I)
2.2.7	Longword Integer Format in Floating-Point Unit	2-10 (I)
2.2.8	Quadword Integer Format in Floating-Point Unit	2-11 (I)
2.2.9	Data Types with No Hardware Support	2-11 (I)

2.3	Big-Endian Addressing Support	2-12 (I)
-----	-------------------------------------	----------

3 Instruction Formats (I)

3.1	Alpha Registers	3-1 (I)
3.1.1	Program Counter	3-1 (I)
3.1.2	Integer Registers	3-1 (I)
3.1.3	Floating-Point Registers	3-2 (I)
3.1.4	Lock Registers	3-2 (I)
3.1.5	Processor Cycle Counter (PCC) Register	3-3 (I)
3.1.6	Optional Registers	3-3 (I)
3.1.6.1	Memory Prefetch Registers	3-3 (I)
3.1.6.2	VAX Compatibility Register	3-3 (I)
3.2	Notation	3-3 (I)
3.2.1	Operand Notation	3-4 (I)
3.2.2	Instruction Operand Notation	3-4 (I)
3.2.2.1	Operand Name Notation	3-5 (I)
3.2.2.2	Operand Access Type Notation	3-5 (I)
3.2.2.3	Operand Data Type Notation	3-6 (I)
3.2.3	Operators	3-6 (I)
3.2.4	Notation Conventions	3-9 (I)
3.3	Instruction Formats	3-9 (I)
3.3.1	Memory Instruction Format	3-10 (I)
3.3.1.1	Memory Format Instructions with a Function Code	3-10 (I)
3.3.1.2	Memory Format Jump Instructions	3-10 (I)
3.3.2	Branch Instruction Format	3-11 (I)
3.3.3	Operate Instruction Format	3-11 (I)
3.3.4	Floating-Point Operate Instruction Format	3-12 (I)
3.3.4.1	Floating-Point Convert Instructions	3-13 (I)
3.3.4.2	Floating-Point/Integer Register Moves	3-13 (I)
3.3.5	PALcode Instruction Format	3-13 (I)

4 Instruction Descriptions (I)

4.1	Instruction Set Overview	4-1 (I)
4.1.1	Subsetting Rules	4-2 (I)
4.1.2	Floating-Point Subsets	4-2 (I)
4.1.3	Software Emulation Rules	4-2 (I)
4.1.4	Opcode Qualifiers	4-3 (I)
4.2	Memory Integer Load/Store Instructions	4-4 (I)
4.2.1	Load Address	4-5 (I)
4.2.2	Load Memory Data into Integer Register	4-6 (I)
4.2.3	Load Unaligned Memory Data into Integer Register	4-8 (I)
4.2.4	Load Memory Data into Integer Register Locked	4-9 (I)
4.2.5	Store Integer Register Data into Memory Conditional	4-13 (I)
4.2.6	Store Integer Register Data into Memory	4-16 (I)
4.2.7	Store Unaligned Integer Register Data into Memory	4-18 (I)
4.3	Control Instructions	4-19 (I)
4.3.1	Conditional Branch	4-21 (I)
4.3.2	Unconditional Branch	4-22 (I)
4.3.3	Jumps	4-23 (I)
4.4	Integer Arithmetic Instructions	4-25 (I)
4.4.1	Longword Add	4-26 (I)
4.4.2	Scaled Longword Add	4-27 (I)
4.4.3	Quadword Add	4-28 (I)
4.4.4	Scaled Quadword Add	4-29 (I)

4.4.5	Integer Signed Compare	4-30 (I)
4.4.6	Integer Unsigned Compare	4-31 (I)
4.4.7	Count Leading Zero	4-32 (I)
4.4.8	Count Population	4-33 (I)
4.4.9	Count Trailing Zero	4-34 (I)
4.4.10	Longword Multiply	4-35 (I)
4.4.11	Quadword Multiply	4-36 (I)
4.4.12	Unsigned Quadword Multiply High	4-37 (I)
4.4.13	Longword Subtract	4-38 (I)
4.4.14	Scaled Longword Subtract	4-39 (I)
4.4.15	Quadword Subtract	4-40 (I)
4.4.16	Scaled Quadword Subtract	4-41 (I)
4.5	Logical and Shift Instructions	4-42 (I)
4.5.1	Logical Functions	4-43 (I)
4.5.2	Conditional Move Integer	4-44 (I)
4.5.3	Shift Logical	4-46 (I)
4.5.4	Shift Arithmetic	4-47 (I)
4.6	Byte Manipulation Instructions	4-48 (I)
4.6.1	Compare Byte	4-50 (I)
4.6.2	Extract Byte	4-52 (I)
4.6.3	Byte Insert	4-56 (I)
4.6.4	Byte Mask	4-58 (I)
4.6.5	Sign Extend	4-61 (I)
4.6.6	Zero Bytes	4-62 (I)
4.7	Floating-Point Instructions	4-63 (I)
4.7.1	Single-Precision Operations	4-63 (I)
4.7.2	Subsets and Faults	4-63 (I)
4.7.3	Definitions	4-64 (I)
4.7.4	Encodings	4-66 (I)
4.7.5	Rounding Modes	4-67 (I)
4.7.6	Computational Models	4-68 (I)
4.7.6.1	VAX-Format Arithmetic with Precise Exceptions	4-68 (I)
4.7.6.2	High-Performance VAX-Format Arithmetic	4-68 (I)
4.7.6.3	IEEE-Compliant Arithmetic	4-69 (I)
4.7.6.4	IEEE-Compliant Arithmetic Without Inexact Exception	4-69 (I)
4.7.6.5	High-Performance IEEE-Format Arithmetic	4-69 (I)
4.7.7	Trapping Modes	4-70 (I)
4.7.7.1	VAX Trapping Modes	4-70 (I)
4.7.7.2	IEEE Trapping Modes	4-72 (I)
4.7.7.3	Arithmetic Trap Completion	4-73 (I)
4.7.7.3.1	Trap Shadow Rules	4-74 (I)
4.7.7.3.2	Trap Shadow Length Rules	4-75 (I)
4.7.7.4	Invalid Operation (INV) Arithmetic Trap	4-77 (I)
4.7.7.5	Division by Zero (DZE) Arithmetic Trap	4-78 (I)
4.7.7.6	Overflow (OVF) Arithmetic Trap	4-78 (I)
4.7.7.7	Underflow (UNF) Arithmetic Trap	4-78 (I)
4.7.7.8	Inexact Result (INE) Arithmetic Trap	4-78 (I)
4.7.7.9	Integer Overflow (IOV) Arithmetic Trap	4-79 (I)
4.7.7.10	IEEE Floating-Point Trap Disable Bits	4-79 (I)
4.7.7.11	IEEE Denormal Control Bits	4-79 (I)
4.7.8	Floating-Point Control Register (FPCR)	4-80 (I)
4.7.8.1	Accessing the FPCR	4-82 (I)
4.7.8.2	Default Values of the FPCR	4-83 (I)
4.7.8.3	Saving and Restoring the FPCR	4-83 (I)
4.7.9	Floating-Point Instruction Function Field Format	4-84 (I)
4.7.10	IEEE Standard	4-89 (I)
4.7.10.1	Conversion of NaN and Infinity Values	4-89 (I)
4.7.10.2	Copying NaN Values	4-89 (I)
4.7.10.3	Generating NaN Values	4-89 (I)

4.7.10.4	Propagating NaN Values	4-89 (I)
4.8	Memory Format Floating-Point Instructions	4-90 (I)
4.8.1	Load F_floating	4-91 (I)
4.8.2	Load G_floating	4-92 (I)
4.8.3	Load S_floating	4-93 (I)
4.8.4	Load T_floating	4-94 (I)
4.8.5	Store F_floating	4-95 (I)
4.8.6	Store G_floating	4-96 (I)
4.8.7	Store S_floating	4-97 (I)
4.8.8	Store T_floating	4-98 (I)
4.9	Branch Format Floating-Point Instructions	4-99 (I)
4.9.1	Conditional Branch	4-100 (I)
4.10	Floating-Point Operate Format Instructions	4-101 (I)
4.10.1	Copy Sign	4-104 (I)
4.10.2	Convert Integer to Integer	4-105 (I)
4.10.3	Floating-Point Conditional Move	4-106 (I)
4.10.4	Move from/to Floating-Point Control Register	4-108 (I)
4.10.5	VAX Floating Add	4-109 (I)
4.10.6	IEEE Floating Add	4-110 (I)
4.10.7	VAX Floating Compare	4-111 (I)
4.10.8	IEEE Floating Compare	4-112 (I)
4.10.9	Convert VAX Floating to Integer	4-113 (I)
4.10.10	Convert Integer to VAX Floating	4-114 (I)
4.10.11	Convert VAX Floating to VAX Floating	4-115 (I)
4.10.12	Convert IEEE Floating to Integer	4-116 (I)
4.10.13	Convert Integer to IEEE Floating	4-117 (I)
4.10.14	Convert IEEE S_floating to IEEE T_floating	4-118 (I)
4.10.15	Convert IEEE T_floating to IEEE S_floating	4-119 (I)
4.10.16	VAX Floating Divide	4-120 (I)
4.10.17	IEEE Floating Divide	4-121 (I)
4.10.18	Floating-Point Register to Integer Register Move	4-122 (I)
4.10.19	Integer Register to Floating-Point Register Move	4-124 (I)
4.10.20	VAX Floating Multiply	4-126 (I)
4.10.21	IEEE Floating Multiply	4-127 (I)
4.10.22	VAX Floating Square Root	4-128 (I)
4.10.23	IEEE Floating Square Root	4-129 (I)
4.10.24	VAX Floating Subtract	4-130 (I)
4.10.25	IEEE Floating Subtract	4-131 (I)
4.11	Miscellaneous Instructions	4-132 (I)
4.11.1	Architecture Mask	4-133 (I)
4.11.2	Call Privileged Architecture Library	4-135 (I)
4.11.3	Evict Data Cache Block	4-136 (I)
4.11.4	Exception Barrier	4-138 (I)
4.11.5	Prefetch Data	4-139 (I)
4.11.6	Implementation Version	4-141 (I)
4.11.7	Memory Barrier	4-142 (I)
4.11.8	Prefetch Memory Data	4-143 (I)
4.11.9	Read Processor Cycle Counter	4-145 (I)
4.11.10	Trap Barrier	4-147 (I)
4.11.11	Write Hint	4-148 (I)
4.11.12	Write Memory Barrier	4-150 (I)
4.12	VAX Compatibility Instructions	4-152 (I)
4.12.1	VAX Compatibility Instructions	4-153 (I)
4.13	Multimedia (Graphics and Video) Support	4-154 (I)
4.13.1	Byte and Word Minimum and Maximum	4-155 (I)
4.13.2	Pixel Error	4-157 (I)
4.13.3	Pack Bytes	4-158 (I)
4.13.4	Unpack Bytes	4-159 (I)

5 System Architecture and Programming Implications (I)

5.1	Introduction	5-1 (I)
5.2	Physical Address Space Characteristics	5-1 (I)
5.2.1	Coherency of Memory Access	5-1 (I)
5.2.2	Granularity of Memory Access	5-2 (I)
5.2.3	Width of Memory Access	5-3 (I)
5.2.4	Memory-Like and Non-Memory-Like Behavior	5-3 (I)
5.3	Translation Buffers and Virtual Caches	5-4 (I)
5.4	Caches and Write Buffers	5-4 (I)
5.5	Data Sharing	5-6 (I)
5.5.1	Atomic Change of a Single Datum	5-6 (I)
5.5.2	Atomic Update of a Single Datum	5-6 (I)
5.5.3	Atomic Update of Data Structures	5-7 (I)
5.5.4	Prefetching Low-Contention Atomic Data and Locks	5-8 (I)
5.5.5	Ordering Considerations for Shared Data Structures	5-9 (I)
5.6	Read/Write Ordering	5-10 (I)
5.6.1	Alpha Shared Memory Model	5-11 (I)
5.6.1.1	Architectural Definition of Processor Issue Sequence	5-12 (I)
5.6.1.2	Definition of Before and After	5-13 (I)
5.6.1.3	Definition of Processor Issue Constraints	5-13 (I)
5.6.1.4	Definition of Location Access Constraints	5-14 (I)
5.6.1.5	Definition of Visibility	5-14 (I)
5.6.1.6	Definition of Storage	5-15 (I)
5.6.1.7	Definition of Dependence Constraint	5-15 (I)
5.6.1.8	Definition of Load-Locked and Store-Conditional	5-16 (I)
5.6.1.9	Timeliness	5-17 (I)
5.6.2	Litmus Tests	5-17 (I)
5.6.2.1	Litmus Test 1 (Impossible Sequence)	5-17 (I)
5.6.2.2	Litmus Test 2 (Impossible Sequence)	5-18 (I)
5.6.2.3	Litmus Test 3 (Impossible Sequence)	5-18 (I)
5.6.2.4	Litmus Test 4 (Sequence Okay)	5-19 (I)
5.6.2.5	Litmus Test 5 (Sequence Okay)	5-19 (I)
5.6.2.6	Litmus Test 6 (Sequence Okay)	5-20 (I)
5.6.2.7	Litmus Test 7 (Impossible Sequence)	5-20 (I)
5.6.2.8	Litmus Test 8 (Impossible Sequence)	5-21 (I)
5.6.2.9	Litmus Test 9 (Impossible Sequence)	5-21 (I)
5.6.2.10	Litmus Test 10 (Sequence Okay)	5-22 (I)
5.6.2.11	Litmus Test 11 (Impossible Sequence)	5-22 (I)
5.6.3	Implied Barriers	5-22 (I)
5.6.4	Implications for Software	5-23 (I)
5.6.4.1	Single Processor Data Stream	5-23 (I)
5.6.4.2	Single Processor Instruction Stream	5-23 (I)
5.6.4.3	Multiprocessor Data Stream (Including Single Processor with DMA I/O)	5-23 (I)
5.6.4.4	Multiprocessor Instruction Stream (Including Single Processor with DMA I/O)	5-24 (I)
5.6.4.5	Multiprocessor Context Switch	5-24 (I)
5.6.4.6	Multiprocessor Send/Receive Interrupt	5-26 (I)
5.6.4.7	Implications for Memory Mapped I/O	5-27 (I)
5.6.4.8	Multiple Processors Writing to a Single I/O Device	5-28 (I)
5.6.5	Implications for Hardware	5-28 (I)
5.7	Arithmetic Traps	5-29 (I)

6 Common PALcode Architecture (I)

6.1	PALcode	6-1 (I)
6.2	PALcode Instructions and Functions	6-1 (I)
6.3	PALcode Environment	6-2 (I)

6.4	Special Functions Required for PALcode	6-2 (I)
6.5	PALcode Effects on System Code	6-3 (I)
6.6	PALcode Replacement	6-3 (I)
6.7	Required PALcode Instructions	6-4 (I)
6.7.1	Drain Aborts	6-5 (I)
6.7.2	Halt	6-6 (I)
6.7.3	Instruction Memory Barrier	6-7 (I)

7 Console Subsystem Overview (I)

8 Input/Output Overview (I)

9 Introduction to OpenVMS (II-A)

9.1	Register Usage	9-1 (II-A)
9.1.1	Processor Status	9-1 (II-A)
9.1.2	Stack Pointer (SP)	9-1 (II-A)
9.1.3	Internal Processor Registers (IPRs)	9-1 (II-A)
9.1.4	Processor Cycle Counter (PCC)	9-2 (II-A)

10 PALcode Instruction Descriptions (II-A)

10.1	Unprivileged General PALcode Instructions	10-3 (II-A)
10.1.1	Breakpoint	10-4 (II-A)
10.1.2	Bugcheck	10-5 (II-A)
10.1.3	Change Mode to Executive	10-6 (II-A)
10.1.4	Change Mode to Kernel	10-7 (II-A)
10.1.5	Change Mode to Supervisor	10-8 (II-A)
10.1.6	Change Mode to User	10-9 (II-A)
10.1.7	Clear Floating-Point Enable	10-10 (II-A)
10.1.8	Generate Software Trap	10-11 (II-A)
10.1.9	Probe Memory Access	10-12 (II-A)
10.1.10	Read Processor Status	10-13 (II-A)
10.1.11	Return from Exception or Interrupt	10-14 (II-A)
10.1.12	Read System Cycle Counter	10-16 (II-A)
10.1.13	Swap AST Enable	10-18 (II-A)
10.1.14	Write Processor Status Software Field	10-19 (II-A)
10.2	Queue Data Types	10-20 (II-A)
10.2.1	Absolute Longword Queues	10-20 (II-A)
10.2.2	Self-Relative Longword Queues	10-20 (II-A)
10.2.3	Absolute Quadword Queues	10-23 (II-A)
10.2.4	Self-Relative Quadword Queues	10-24 (II-A)
10.3	Unprivileged Queue PALcode Instructions	10-28 (II-A)
10.3.1	Insert Entry into Longword Queue at Head Interlocked	10-29 (II-A)
10.3.2	Insert Entry into Longword Queue at Head Interlocked Resident	10-31 (II-A)
10.3.3	Insert Entry into Quadword Queue at Head Interlocked	10-33 (II-A)
10.3.4	Insert Entry into Quadword Queue at Head Interlocked Resident	10-35 (II-A)
10.3.5	Insert Entry into Longword Queue at Tail Interlocked	10-37 (II-A)
10.3.6	Insert Entry into Longword Queue at Tail Interlocked Resident	10-39 (II-A)
10.3.7	Insert Entry into Quadword Queue at Tail Interlocked	10-41 (II-A)
10.3.8	Insert Entry into Quadword Queue at Tail Interlocked Resident	10-43 (II-A)
10.3.9	Insert Entry into Longword Queue	10-45 (II-A)
10.3.10	Insert Entry into Quadword Queue	10-47 (II-A)
10.3.11	Remove Entry from Longword Queue at Head Interlocked	10-49 (II-A)

10.3.12	Remove Entry from Longword Queue at Head Interlocked Resident	10-52 (II-A)
10.3.13	Remove Entry from Quadword Queue at Head Interlocked	10-54 (II-A)
10.3.14	Remove Entry from Quadword Queue at Head Interlocked Resident	10-57 (II-A)
10.3.15	Remove Entry from Longword Queue at Tail Interlocked	10-59 (II-A)
10.3.16	Remove Entry from Longword Queue at Tail Interlocked Resident	10-62 (II-A)
10.3.17	Remove Entry from Quadword Queue at Tail Interlocked	10-64 (II-A)
10.3.18	Remove Entry from Quadword Queue at Tail Interlocked Resident	10-67 (II-A)
10.3.19	Remove Entry from Longword Queue	10-69 (II-A)
10.3.20	Remove Entry from Quadword Queue	10-71 (II-A)
10.4	Unprivileged VAX Compatibility PALcode Instructions	10-73 (II-A)
10.4.1	Atomic Move Operation	10-74 (II-A)
10.5	Unprivileged PALcode Thread Instructions	10-78 (II-A)
10.5.1	Read Unique Context	10-79 (II-A)
10.5.2	Write Unique Context	10-80 (II-A)
10.6	Privileged PALcode Instructions	10-81 (II-A)
10.6.1	Cache Flush	10-82 (II-A)
10.6.2	Console Service	10-83 (II-A)
10.6.3	Load Quadword Physical	10-84 (II-A)
10.6.4	Move from Processor Register	10-85 (II-A)
10.6.5	Move to Processor Register	10-86 (II-A)
10.6.6	Store Quadword Physical	10-87 (II-A)
10.6.7	Swap Privileged Context	10-88 (II-A)
10.6.8	Swap PALcode Image	10-91 (II-A)
10.6.9	Wait for Interrupt	10-93 (II-A)

11 Memory Management (II-A)

11.1	Introduction	11-1 (II-A)
11.2	Virtual Address Space	11-1 (II-A)
11.3	Virtual Address Format	11-2 (II-A)
11.4	Physical Address Space	11-3 (II-A)
11.5	Memory Management Control	11-3 (II-A)
11.6	Page Table Entries	11-3 (II-A)
11.6.1	Changes to Page Table Entries	11-6 (II-A)
11.7	Memory Protection	11-7 (II-A)
11.7.1	Processor Access Modes	11-7 (II-A)
11.7.2	Protection Code	11-7 (II-A)
11.7.3	Access Violation Fault	11-8 (II-A)
11.8	Address Translation	11-8 (II-A)
11.8.1	Physical Access for Page Table Entries	11-8 (II-A)
11.8.2	Virtual Access for Page Table Entries	11-10 (II-A)
11.8.3	Reduced Page Table (RPT) Mode	11-11 (II-A)
11.8.3.1	Physical Access for Page Table Entries in Reduced Page Table Mode	11-12 (II-A)
11.8.3.2	Virtual Access for Page Table Entries in Reduced Page Table Mode	11-13 (II-A)
11.9	Translation Buffer	11-13 (II-A)
11.10	Address Space Numbers	11-14 (II-A)
11.11	Memory Management Faults	11-15 (II-A)

12 Process Structure (II-A)

12.1	Process Definition	12-1 (II-A)
12.2	Hardware Privileged Process Context	12-2 (II-A)
12.3	Asynchronous System Traps (AST)	12-3 (II-A)
12.4	Process Context Switching	12-4 (II-A)

13 Internal Processor Registers (II–A)

13.1	Internal Processor Registers	13–1 (II-A)
13.2	Stack Pointer Internal Processor Registers	13–1 (II-A)
13.3	IPR Summary	13–2 (II-A)
13.3.1	Address Space Number (ASN)	13–4 (II-A)
13.3.2	AST Enable (ASTEN)	13–5 (II-A)
13.3.3	AST Summary Register (ASTSR)	13–7 (II-A)
13.3.4	Data Alignment Trap Fixup (DATFX)	13–9 (II-A)
13.3.5	Executive Stack Pointer (ESP)	13–10 (II-A)
13.3.6	Floating Enable (FEN)	13–11 (II-A)
13.3.7	Interprocessor Interrupt Request (IPIR)	13–12 (II-A)
13.3.8	Interrupt Priority Level (IPL)	13–13 (II-A)
13.3.9	Machine Check Error Summary Register (MCES)	13–14 (II-A)
13.3.10	Performance Monitoring Register (PERFMON)	13–15 (II-A)
13.3.11	Privileged Context Block Base (PCBB)	13–16 (II-A)
13.3.12	Processor Base Register (PRBR)	13–17 (II-A)
13.3.13	Page Table Base Register (PTBR)	13–18 (II-A)
13.3.14	System Control Block Base (SCBB)	13–19 (II-A)
13.3.15	Software Interrupt Request Register (SIRR)	13–20 (II-A)
13.3.16	Software Interrupt Summary Register (SISR)	13–21 (II-A)
13.3.17	Supervisor Stack Pointer (SSP)	13–22 (II-A)
13.3.18	System Page Table Base Register (SYSPTBR)	13–23 (II-A)
13.3.19	Translation Buffer Check (TBCHK)	13–24 (II-A)
13.3.20	Translation Buffer Invalidate All (TBIA)	13–25 (II-A)
13.3.21	Translation Buffer Invalidate All Process (TBIAP)	13–26 (II-A)
13.3.22	Translation Buffer Invalidate Single (TBISx)	13–27 (II-A)
13.3.23	User Stack Pointer (USP)	13–28 (II-A)
13.3.24	Virtual Address Boundary Register (VIRBND)	13–29 (II-A)
13.3.25	Virtual Page Table Base (VPTB)	13–30 (II-A)
13.3.26	Who-Am-I (WHAMI)	13–31 (II-A)

14 Exceptions, Interrupts, and Machine Checks (II–A)

14.1	Introduction	14–1 (II-A)
14.1.1	Differences Between Exceptions, Interrupts, and Machine Checks	14–2 (II-A)
14.1.2	Exceptions, Interrupts, and Machine Checks Summary	14–2 (II-A)
14.2	Processor State and Exception/Interrupt/Machine Check Stack Frame	14–4 (II-A)
14.2.1	Processor Status	14–5 (II-A)
14.2.2	Program Counter	14–6 (II-A)
14.2.3	Processor Interrupt Priority Level (IPL)	14–6 (II-A)
14.2.4	Protection Modes	14–7 (II-A)
14.2.5	Processor Stacks	14–7 (II-A)
14.2.6	Stack Frames	14–7 (II-A)
14.3	Exceptions	14–8 (II-A)
14.3.1	Faults	14–9 (II-A)
14.3.1.1	Floating Disabled Fault	14–10 (II-A)
14.3.1.2	Access Control Violation (ACV) Fault	14–10 (II-A)
14.3.1.3	Translation Not Valid (TNV)	14–10 (II-A)
14.3.1.4	Fault on Read (FOR)	14–10 (II-A)
14.3.1.5	Fault on Write (FOW)	14–10 (II-A)
14.3.1.6	Fault on Execute (FOE)	14–11 (II-A)
14.3.2	Arithmetic Traps	14–11 (II-A)
14.3.2.1	Exception Summary Parameter	14–12 (II-A)
14.3.2.2	Register Write Mask	14–13 (II-A)
14.3.2.3	Invalid Operation (INV) Trap	14–13 (II-A)
14.3.2.4	Division by Zero (DZE) Trap	14–13 (II-A)

14.3.2.5	Overflow (OVF) Trap	14–13 (II-A)
14.3.2.6	Underflow (UNF) Trap	14–13 (II-A)
14.3.2.7	Inexact Result (INE) Trap	14–13 (II-A)
14.3.2.8	Integer Overflow (IOV) Trap	14–14 (II-A)
14.3.3	Synchronous Traps	14–14 (II-A)
14.3.3.1	Data Alignment Trap	14–14 (II-A)
14.3.3.2	Other Synchronous Traps	14–15 (II-A)
14.3.3.2.1	Breakpoint Trap	14–15 (II-A)
14.3.3.2.2	Bugcheck Trap	14–15 (II-A)
14.3.3.2.3	Illegal Instruction Trap	14–15 (II-A)
14.3.3.2.4	Illegal Operand Trap	14–15 (II-A)
14.3.3.2.5	Generate Software Trap	14–15 (II-A)
14.3.3.2.6	Change Mode to Kernel Trap	14–16 (II-A)
14.3.3.2.7	Change Mode to Executive Trap	14–16 (II-A)
14.3.3.2.8	Change Mode to Supervisor Trap	14–16 (II-A)
14.3.3.2.9	Change Mode to User Trap	14–16 (II-A)
14.4	Interrupts	14–16 (II-A)
14.4.1	Software Interrupts — IPLs 1 to 15	14–17 (II-A)
14.4.1.1	Software Interrupt Summary Register	14–17 (II-A)
14.4.1.2	Software Interrupt Request Register	14–18 (II-A)
14.4.2	Asynchronous System Trap — IPL 2	14–18 (II-A)
14.4.3	Passive Release Interrupts — IPLs 20 to 23	14–19 (II-A)
14.4.4	I/O Device Interrupts — IPLs 20 to 23	14–19 (II-A)
14.4.5	Interval Clock Interrupt — IPL 22	14–19 (II-A)
14.4.6	Interprocessor Interrupt — IPL 22	14–19 (II-A)
14.4.6.1	Interprocessor Interrupt Request Register	14–19 (II-A)
14.4.7	Performance Monitor Interrupts — IPL 29	14–20 (II-A)
14.4.8	Powerfail Interrupt — IPL 30	14–20 (II-A)
14.5	Machine Checks	14–21 (II-A)
14.5.1	Software Response	14–22 (II-A)
14.5.2	Logout Areas	14–23 (II-A)
14.6	System Control Block	14–24 (II-A)
14.6.1	SCB Entries for Faults	14–25 (II-A)
14.6.2	SCB Entries for Arithmetic Traps	14–25 (II-A)
14.6.3	SCB Entries for Asynchronous System Traps (ASTs)	14–26 (II-A)
14.6.4	SCB Entries for Data Alignment Traps	14–26 (II-A)
14.6.5	SCB Entries for Other Synchronous Traps	14–26 (II-A)
14.6.6	SCB Entries for Processor Software Interrupts	14–27 (II-A)
14.6.7	SCB Entries for Processor Hardware Interrupts and Machine Checks	14–27 (II-A)
14.6.8	SCB Entries for I/O Device Interrupts	14–28 (II-A)
14.7	PALcode Support	14–28 (II-A)
14.7.1	Stack Writeability	14–28 (II-A)
14.7.2	Stack Residency	14–28 (II-A)
14.7.3	Stack Alignment	14–29 (II-A)
14.7.4	Initiate Exception or Interrupt or Machine Check	14–29 (II-A)
14.7.5	Initiate Exception or Interrupt or Machine Check Model	14–29 (II-A)
14.7.6	PALcode Interrupt Arbitration	14–32 (II-A)
14.7.6.1	Writing the AST Summary Register	14–32 (II-A)
14.7.6.2	Writing the AST Enable Register	14–32 (II-A)
14.7.6.3	Writing the IPL Register	14–32 (II-A)
14.7.6.4	Writing the Software Interrupt Request Register	14–33 (II-A)
14.7.6.4.1	Return from Exception or Interrupt	14–33 (II-A)
14.7.6.5	Swap AST Enable	14–34 (II-A)
14.7.7	Processor State Transition Table	14–34 (II-A)

15 Introduction to Tru64 UNIX (II-B)

15.1	Programming Model	15-2 (II-B)
15.1.1	Code Flow Constants and Terms	15-2 (II-B)
15.1.2	Machine State Terms	15-2 (II-B)

16 PALcode Instruction Descriptions (II-B)

16.1	Unprivileged PALcode Instructions	16-1 (II-B)
16.1.1	Breakpoint Trap	16-2 (II-B)
16.1.2	Bugcheck Trap	16-3 (II-B)
16.1.3	System Call	16-4 (II-B)
16.1.4	Clear Floating-Point Enable	16-5 (II-B)
16.1.5	Generate Trap	16-6 (II-B)
16.1.6	Read Unique Value	16-7 (II-B)
16.1.7	Return from User Mode Trap	16-8 (II-B)
16.1.8	Write Unique Value	16-9 (II-B)
16.2	Privileged PALcode Instructions	16-10 (II-B)
16.2.1	Cache Flush	16-11 (II-B)
16.2.2	Console Service	16-12 (II-B)
16.2.3	Read Machine Check Error Summary	16-13 (II-B)
16.2.4	Read Processor Status	16-14 (II-B)
16.2.5	Read User Stack Pointer	16-15 (II-B)
16.2.6	Read System Value	16-16 (II-B)
16.2.7	Return from System Call	16-17 (II-B)
16.2.8	Return from Trap, Fault or Interrupt	16-18 (II-B)
16.2.9	Swap Process Context	16-19 (II-B)
16.2.10	Swap IPL	16-21 (II-B)
16.2.11	Swap PALcode Image	16-22 (II-B)
16.2.12	TB Invalidate	16-24 (II-B)
16.2.13	Who Am I	16-25 (II-B)
16.2.14	Write ASN	16-26 (II-B)
16.2.15	Write System Entry Address	16-27 (II-B)
16.2.16	Write Floating-Point Enable	16-28 (II-B)
16.2.17	Write Interprocessor Interrupt Request	16-29 (II-B)
16.2.18	Write Kernel Global Pointer	16-30 (II-B)
16.2.19	Write Machine Check Error Summary	16-31 (II-B)
16.2.20	Performance Monitoring Function	16-32 (II-B)
16.2.21	Write System Page Table Base	16-33 (II-B)
16.2.22	Write User Stack Pointer	16-34 (II-B)
16.2.23	Write System Value	16-35 (II-B)
16.2.24	Write Virtual Address Boundary	16-36 (II-B)
16.2.25	Write Virtual Page Table Pointer	16-37 (II-B)
16.2.26	Wait For Interrupt	16-38 (II-B)

17 Memory Management (II-B)

17.1	Virtual Address Spaces	17-1 (II-B)
17.1.1	Segment Seg0 and Seg1 Virtual Address Format	17-2 (II-B)
17.1.2	Kseg Virtual Address Format	17-3 (II-B)
17.2	Physical Address Space	17-3 (II-B)
17.3	Memory Management Control	17-3 (II-B)
17.4	Page Table Entries	17-3 (II-B)
17.4.1	Changes to Page Table Entries	17-6 (II-B)
17.5	Memory Protection	17-6 (II-B)
17.5.1	Processor Access Modes	17-7 (II-B)

17.5.2	Protection Code	17-7 (II-B)
17.5.3	Access-Violation Faults	17-7 (II-B)
17.6	Address Translation for Seg0 and Seg1	17-7 (II-B)
17.6.1	Physical Access for Seg0 and Seg1 PTEs	17-7 (II-B)
17.6.2	Virtual Access for Seg0 or Seg1 PTEs	17-9 (II-B)
17.6.3	Reduced Page Table (RPT) Mode	17-10 (II-B)
17.6.3.1	Physical Access for Page Table Entries in Reduced Page Table Mode	17-10 (II-B)
17.6.3.2	Virtual Access for Page Table Entries in Reduced Page Table Mode	17-12 (II-B)
17.7	Translation Buffer	17-12 (II-B)
17.8	Address Space Numbers	17-12 (II-B)
17.9	Memory-Management Faults	17-13 (II-B)

18 Process Structure (II-B)

18.1	Process Definition	18-1 (II-B)
18.2	Process Control Block (PCB)	18-2 (II-B)

19 Exceptions and Interrupts (II-B)

19.1	Introduction	19-1 (II-B)
19.1.1	Exceptions	19-1 (II-B)
19.1.2	Interrupts	19-2 (II-B)
19.2	Processor Status	19-2 (II-B)
19.3	Stack Frames	19-3 (II-B)
19.4	System Entry Addresses	19-4 (II-B)
19.4.1	System Entry Arithmetic Trap (entArith)	19-4 (II-B)
19.4.1.1	Exception Summary Register	19-4 (II-B)
19.4.1.2	Exception Register Write Mask	19-6 (II-B)
19.4.2	System Entry Instruction Fault (entIF)	19-6 (II-B)
19.4.3	System Entry Hardware Interrupts (entInt)	19-7 (II-B)
19.4.4	System Entry MM Fault (entMM)	19-8 (II-B)
19.4.5	System Entry Call System (entSys)	19-9 (II-B)
19.4.6	System Entry Unaligned Access (entUna)	19-9 (II-B)
19.5	PALcode Support	19-9 (II-B)
19.5.1	Stack Writeability and Alignment	19-9 (II-B)

20 Introduction to Alpha Linux (II-C)

20.1	Programming Model	20-2 (II-C)
20.1.1	Code Flow Constants and Terms	20-2 (II-C)
20.1.2	Machine State Terms	20-2 (II-C)

21 PALcode Instruction Descriptions (II-C)

21.1	Unprivileged PALcode Instructions	21-1 (II-C)
21.1.1	Breakpoint Trap	21-2 (II-C)
21.1.2	Bugcheck Trap	21-3 (II-C)
21.1.3	System Call	21-4 (II-C)
21.1.4	Clear Floating-Point Enable	21-5 (II-C)
21.1.5	Generate Trap	21-6 (II-C)
21.1.6	Read Unique Value	21-7 (II-C)
21.1.7	Write Unique Value	21-8 (II-C)
21.2	Privileged PALcode Instructions	21-9 (II-C)
21.2.1	Cache Flush	21-10 (II-C)

21.2.2	Console Service	21-11 (II-C)
21.2.3	Read Machine Check Error Summary	21-12 (II-C)
21.2.4	Read Processor Status	21-13 (II-C)
21.2.5	Read User Stack Pointer	21-14 (II-C)
21.2.6	Read System Value	21-15 (II-C)
21.2.7	Return from System Call	21-16 (II-C)
21.2.8	Return from Trap, Fault or Interrupt	21-17 (II-C)
21.2.9	Swap Process Context	21-18 (II-C)
21.2.10	Swap IPL	21-20 (II-C)
21.2.11	Swap PALcode Image	21-21 (II-C)
21.2.12	TB Invalidate	21-23 (II-C)
21.2.13	Who Am I	21-24 (II-C)
21.2.14	Write ASN	21-25 (II-C)
21.2.15	Write System Entry Address	21-26 (II-C)
21.2.16	Write Floating-Point Enable	21-27 (II-C)
21.2.17	Write Interprocessor Interrupt Request	21-28 (II-C)
21.2.18	Write Kernel Global Pointer	21-29 (II-C)
21.2.19	Write Machine Check Error Summary	21-30 (II-C)
21.2.20	Performance Monitoring Function	21-31 (II-C)
21.2.21	Write System Page Table Base	21-32 (II-C)
21.2.22	Write User Stack Pointer	21-33 (II-C)
21.2.23	Write System Value	21-34 (II-C)
21.2.24	Write Virtual Address Boundary	21-35 (II-C)
21.2.25	Write Virtual Page Table Pointer	21-36 (II-C)
21.2.26	Wait for Interrupt	21-37 (II-C)

22 Memory Management (II-C)

22.1	Virtual Address Spaces	22-1 (II-C)
22.1.1	Segment Seg0 and Seg1 Virtual Address Format	22-2 (II-C)
22.1.2	Kseg Virtual Address Format	22-3 (II-C)
22.2	Physical Address Space	22-3 (II-C)
22.3	Memory Management Control	22-3 (II-C)
22.4	Page Table Entries	22-3 (II-C)
22.4.1	Changes to Page Table Entries	22-6 (II-C)
22.5	Memory Protection	22-6 (II-C)
22.5.1	Processor Access Modes	22-7 (II-C)
22.5.2	Protection Code	22-7 (II-C)
22.5.3	Access-Violation Faults	22-7 (II-C)
22.6	Address Translation for Seg0 and Seg1	22-7 (II-C)
22.6.1	Physical Access for Seg0 and Seg1 PTEs	22-7 (II-C)
22.6.2	Virtual Access for Seg0 or Seg1 PTEs	22-9 (II-C)
22.6.3	Reduced Page Table (RPT) Mode	22-10 (II-C)
22.6.3.1	Physical Access for Page Table Entries in Reduced Page Table Mode	22-10 (II-C)
22.6.3.2	Virtual Access for Page Table Entries in Reduced Page Table Mode	22-12 (II-C)
22.7	Translation Buffer	22-12 (II-C)
22.8	Address Space Numbers	22-12 (II-C)
22.9	Memory-Management Faults	22-13 (II-C)

23 Process Structure (II-C)

23.1	Process Definition	23-1 (II-C)
23.2	Process Control Block (PCB)	23-2 (II-C)

24 Exceptions and Interrupts (II–C)

24.1	Introduction	24–1 (II-C)
24.1.1	Exceptions	24–1 (II-C)
24.1.2	Interrupts	24–2 (II-C)
24.2	Processor Status	24–2 (II-C)
24.3	Stack Frames	24–3 (II-C)
24.4	System Entry Addresses	24–4 (II-C)
24.4.1	System Entry Arithmetic Trap (entArith)	24–4 (II-C)
24.4.1.1	Exception Summary Register	24–4 (II-C)
24.4.1.2	Exception Register Write Mask	24–6 (II-C)
24.4.2	System Entry Instruction Fault (entIF)	24–7 (II-C)
24.4.3	System Entry Hardware Interrupts (entInt)	24–7 (II-C)
24.4.4	System Entry MM Fault (entMM)	24–8 (II-C)
24.4.5	System Entry Call System (entSys)	24–9 (II-C)
24.4.6	System Entry Unaligned Access (entUna)	24–9 (II-C)
24.5	PALcode Support	24–9 (II-C)
24.5.1	Stack Writeability and Alignment	24–9 (II-C)

25 Console Subsystem Overview (III)

25.1	Console Implementations	25–2 (III)
25.2	Console Implementation Registry	25–3 (III)
25.3	Console Presentation Layer	25–3 (III)
25.4	Messages	25–4 (III)
25.5	Security	25–4 (III)
25.6	Internationalization	25–4 (III)
25.7	Documentation Note	25–5 (III)

26 Console Interface to Operating System Software (III)

26.1	Hardware Restart Parameter Block (HWRPB)	26–1 (III)
26.1.1	Serial Number, Revision, Type, and Variation Fields	26–10 (III)
26.1.1.1	Serial Number and Revision Fields	26–11 (III)
26.1.1.2	System Type and Variation Fields	26–12 (III)
26.1.2	Translation Buffer Hint Block	26–13 (III)
26.1.3	Per-CPU Slots in the HWRPB	26–14 (III)
26.1.4	Configuration Data Block	26–23 (III)
26.1.5	Field Replaceable Unit Table	26–23 (III)
26.2	Environment Variables	26–24 (III)
26.3	Console Callback Routines	26–29 (III)
26.3.1	System Software Use of Console Callback Routines	26–29 (III)
26.3.2	System Software Invocation of Console Callback Routines	26–30 (III)
26.3.3	Console Callback Routine Summary	26–30 (III)
26.3.4	Console Terminal Routines	26–32 (III)
26.3.4.1	GETC — Get Character from Console Terminal	26–34 (III)
26.3.4.2	PUTS — Put Stream to Console Terminal	26–36 (III)
26.3.4.3	RESET_TERM — Reset Console Terminal to Default Parameters	26–38 (III)
26.3.4.4	SET_TERM_INT — Set Console Terminal Interrupts	26–39 (III)
26.3.4.5	SET_TERM_CTL — Set Console Terminal Controls	26–41 (III)
26.3.4.6	PROCESS_KEYCODE — Process and Translates Keycode	26–42 (III)
26.3.4.7	CONSOLE_OPEN — Open Console Terminal	26–44 (III)
26.3.4.8	CONSOLE_CLOSE — Close Terminal	26–45 (III)
26.3.5	Console Generic I/O Device Routines	26–46 (III)
26.3.5.1	OPEN — Open Generic I/O Device for Access	26–48 (III)
26.3.5.2	CLOSE — Close Generic I/O Device for Access	26–50 (III)

26.3.5.3	IOCTL — Perform Device-Specific Operations	26-51 (III)
26.3.5.4	READ — Read Generic I/O Device	26-53 (III)
26.3.5.5	WRITE — Write Generic I/O Device	26-55 (III)
26.3.6	Console Environment Variable Routines	26-57 (III)
26.3.6.1	SET_ENV — Set an Environment Variable	26-58 (III)
26.3.6.2	RESET_ENV — Reset an Environment Variable	26-59 (III)
26.3.6.3	GET_ENV — Get an Environment Variable	26-60 (III)
26.3.6.4	SAVE_ENV — Save Current Environment Variables	26-61 (III)
26.3.7	Miscellaneous Routines	26-63 (III)
26.3.7.1	PSWITCH — Switch Primary Processors	26-63 (III)
26.3.7.2	FIXUP — Fixup Virtual Addresses in Console Routines	26-64 (III)
26.3.7.3	BIOS_EMUL — Run BIOS Emulation Callback	26-65 (III)
26.3.8	Console Callback Routine Data Structures	26-68 (III)
26.3.8.1	Console Routine Block	26-69 (III)
26.3.8.1.1	Console Routine Block Initialization	26-71 (III)
26.3.8.1.2	Console Routine Remapping	26-71 (III)
26.3.8.2	Console Terminal Block Table	26-73 (III)
26.4	Interprocessor Console Communications	26-75 (III)
26.4.1	Interprocessor Console Communications Flags	26-75 (III)
26.4.2	Interprocessor Console Communications Buffer Area	26-77 (III)
26.4.3	Sending a Command to a Secondary	26-77 (III)
26.4.4	Sending a Message to the Primary	26-78 (III)

27 System Bootstrapping (III)

27.1	Processor States and Modes	27-1 (III)
27.1.1	States and State Transitions	27-1 (III)
27.1.2	Major Modes	27-3 (III)
27.2	System Initialization	27-3 (III)
27.3	PALcode Loading and Switching	27-4 (III)
27.3.1	PALcode Loading	27-4 (III)
27.3.2	PALcode Switching	27-5 (III)
27.3.2.1	PALcode Switching Procedure	27-6 (III)
27.3.2.2	Specific PALcode Switching Implementation Information	27-7 (III)
27.3.2.3	Processor State at Exit from PALcode Switching Instruction	27-8 (III)
27.4	System Bootstrapping	27-8 (III)
27.4.1	Cold Bootstrapping in a Uniprocessor Environment	27-8 (III)
27.4.1.1	Memory Sizing and Testing	27-9 (III)
27.4.1.2	Passing Memory Cluster Descriptors to System Software	27-10 (III)
27.4.1.2.1	Static Memory Clusters in the MEMDSC Table	27-10 (III)
27.4.1.2.2	Distributed Memory Cluster Descriptors in the FRU Table	27-12 (III)
27.4.1.3	Bootstrap Address Space	27-17 (III)
27.4.1.4	Bootstrap Flags	27-21 (III)
27.4.1.5	Loading of System Software	27-22 (III)
27.4.1.6	Processor Initialization	27-23 (III)
27.4.1.7	Transfer of Control to System Software	27-24 (III)
27.4.2	Warm Bootstrapping in a Uniprocessor Environment	27-25 (III)
27.4.2.1	HWRPB Location and Validation	27-26 (III)
27.4.3	Multiprocessor Bootstrapping	27-27 (III)
27.4.3.1	Selection of Primary Processor	27-27 (III)
27.4.3.2	Actions of Console	27-27 (III)
27.4.3.3	PALcode Loading on Secondary Processors	27-28 (III)
27.4.3.4	Actions of the Running Primary	27-29 (III)
27.4.3.5	Actions of a Console Secondary	27-29 (III)
27.4.3.6	Bootstrap Flags	27-30 (III)
27.4.4	Addition of a Processor to a Running System	27-30 (III)
27.4.5	System Software Requested Bootstraps	27-31 (III)

27.5	System Restarts	27-31 (III)
27.5.1	Actions of Console	27-31 (III)
27.5.2	Powerfail and Recovery — Uniprocessor	27-32 (III)
27.5.3	Powerfail and Recovery — Multiprocessor	27-33 (III)
27.5.3.1	United Powerfail and Recovery	27-33 (III)
27.5.3.2	Split Powerfail and Recovery	27-34 (III)
27.5.4	Error Halt and Recovery.	27-34 (III)
27.5.5	Operator Requested Crash	27-35 (III)
27.5.6	Primary Switching	27-35 (III)
27.5.6.1	Sequence on an Embedded Console	27-35 (III)
27.5.6.2	Sequence on a Detached Console	27-36 (III)
27.5.7	Transitioning Console Terminal State During HALT/RESTART.	27-37 (III)
27.5.7.1	SAVE_TERM — Save Console Terminal State	27-38 (III)
27.5.7.2	RESTORE_TERM — Restore Console Terminal State	27-39 (III)
27.5.8	Operator Forced Entry to Console I/O Mode	27-39 (III)
27.6	Bootstrap Loading and Image Media Format	27-39 (III)
27.6.1	Disk Bootstrapping	27-40 (III)
27.6.2	Tape Bootstrapping	27-42 (III)
27.6.2.1	Bootstrapping from ANSI-Formatted Tape	27-42 (III)
27.6.2.2	Bootstrapping from Boot-Blocked Tape	27-43 (III)
27.6.3	ROM Bootstrapping	27-44 (III)
27.6.4	Network Bootstrapping.	27-45 (III)
27.7	BB_WATCH	27-46 (III)
27.8	Implementation Considerations	27-47 (III)
27.8.1	Embedded Console	27-47 (III)
27.8.1.1	Multiprocessor Considerations	27-48 (III)
27.8.2	Detached Console	27-49 (III)

A Software Considerations

A.1	Hardware-Software Compact	A-1
A.2	Instruction-Stream Considerations	A-2
A.2.1	Instruction Alignment	A-2
A.2.2	Multiple Instruction Issue — Factor of 3.	A-3
A.2.3	Branch Prediction and Minimizing Branch-Taken — Factor of 3.	A-3
A.2.4	Improving I-Stream Density — Factor of 3	A-5
A.2.5	Instruction Scheduling — Factor of 3.	A-5
A.3	Data-Stream Considerations	A-5
A.3.1	Data Alignment — Factor of 10.	A-5
A.3.2	Shared Data in Multiple Processors — Factor of 3	A-6
A.3.3	Avoiding Cache Conflicts — Factor of 1	A-7
A.3.4	Sequential Read/Write — Factor of 1	A-9
A.3.5	Avoid Replay Traps — Factor of 3	A-9
A.3.6	Prefetching — Factor of 3	A-10
A.4	Code Sequences	A-11
A.4.1	Aligned Byte/Word (Within Register) Memory Accesses	A-11
A.4.2	Division	A-12
A.4.3	Byte Swap	A-13
A.4.4	Stylized Code Forms.	A-13
A.4.4.1	NOP	A-13
A.4.4.2	Clear a Register	A-14
A.4.4.3	Load Literal	A-14
A.4.4.4	Register-to-Register Move.	A-15
A.4.4.5	Negate	A-15
A.4.4.6	NOT	A-15
A.4.4.7	Booleans	A-15
A.4.5	Exception and Trap Barriers	A-15

A.4.6	Pseudo-Operations (Stylized Code Forms)	A-16
A.5	Timing Considerations: Atomic Sequences	A-17

B IEEE Floating-Point Conformance

B.1	Alpha Choices for IEEE Options	B-1
B.2	Alpha Support for OS Completion Handlers	B-2
B.2.1	IEEE Floating-Point Control (FP_C) Quadword	B-4
B.3	Mapping to IEEE Standard	B-6

C Instruction Summary

C.1	Common Architecture Instruction Summary	C-1
C.2	IEEE Floating-Point Instructions	C-7
C.3	VAX Floating-Point Instructions	C-9
C.4	Independent Floating-Point Instructions	C-9
C.5	Opcodes Summary	C-10
C.6	Common Architecture Opcodes in Numerical Order	C-12
C.7	OpenVMS PALcode Instruction Summary	C-16
C.8	Tru64 UNIX PALcode Instruction Summary	C-19
C.9	Alpha Linux PALcode Instruction Summary	C-20
C.10	PALcode Opcodes in Numerical Order	C-21
C.11	Required PALcode Opcodes	C-24
C.12	Opcodes Reserved to PALcode	C-24
C.13	Opcodes Reserved to Compaq	C-25
C.14	Unused Function Code Behavior	C-25
C.15	ASCII Character Set	C-26

D Registered System and Processor Identifiers

D.1	Processor Type Assignments	D-1
D.2	PALcode Variation Assignments	D-4
D.3	Architecture Mask and Implementation Values	D-4

E Waivers and Implementation-Dependent Functionality

E.1	Waivers	E-1
E.1.1	21064, 21066, and 21068 IEEE Divide Instruction Violation	E-1
E.1.2	21064, 21066, and 21068 Write Buffer Violation	E-2
E.1.3	21264 LDx_L/STx_C with WH64 Violation	E-2
E.1.4	21164, 21164A, and 21164PC Operation with RPCC Instruction	E-3
E.1.5	21264/EV6 Behavior on LDx_L/STx_C Synchronization	E-3
E.1.6	21264/EV6 and 21264/EV67 Prefetch and Lock Behavior	E-4
E.2	Implementation-Specific Functionality	E-5
E.2.1	Enlarging the Tru64 UNIX kseg Region	E-5
E.2.2	Reduced Page Table (RPT) Mode in the 21364	E-6
E.2.3	21064/21066/21068 Performance Monitoring	E-7
E.2.3.1	21064/21066/21068 Performance Monitor Interrupt Mechanism	E-8
E.2.3.2	Functions and Arguments for the 21064/21066/21068	E-9
E.2.4	21164/21164PC Performance Monitoring	E-12
E.2.4.1	Performance Monitor Interrupt Mechanism	E-12
E.2.4.2	Functions and Arguments	E-13
E.2.5	21264 and 21364 Performance Monitoring	E-22
E.2.5.1	Performance Monitor Interrupt Mechanism	E-22

E.2.5.2	Functions and Arguments	E-23
---------	-----------------------------------	------

F Windows NT Software

F.1	Introduction to Windows NT Alpha Software	F-1
F.1.1	Overview of System Components	F-2
F.1.2	Calling Standard Register Usage	F-4
F.1.3	Code Flow Conventions	F-4
F.2	Processor, Process, Threads, and Registers	F-5
F.2.1	Processor Status	F-5
F.2.2	Internal Processor Register Summary	F-6
F.2.3	Internal Processor Registers	F-7
F.2.4	Processor Data Areas	F-9
F.2.4.1	Processor Control Region	F-9
F.2.4.2	PALcode Version Control	F-10
F.2.4.3	PALcode Alignment Fixup Count	F-10
F.2.5	Caches and Cache Coherency	F-10
F.2.6	Stacks	F-10
F.2.7	Processes and Threads	F-11
F.2.7.1	Swapping Thread Context to Another Thread	F-11
F.2.7.2	Swapping Thread Context to Another Process	F-12
F.3	Memory Management	F-13
F.3.1	Virtual Address Space	F-13
F.3.2	I/O Space Address Extension	F-13
F.3.3	Canonical Virtual Address Format	F-14
F.3.4	Page Table Entries	F-14
F.3.4.1	Single-Level Virtual Traversal of the Page Tables	F-14
F.3.4.2	Two-Level Physical Traversal of the Page Tables	F-15
F.3.4.3	Page Table Entry Summary	F-16
F.3.5	Translation Buffer Management	F-17
F.3.6	Implications of Recursive TB Mapping	F-18
F.4	Exceptions, Interrupts, and Machine Checks	F-19
F.4.1	Exceptions	F-20
F.4.1.1	Exception Dispatch	F-20
F.4.1.2	Exception Classes	F-20
F.4.1.3	Returning from Exceptions	F-21
F.4.1.4	Trap Frames	F-21
F.4.1.5	Memory Management Exceptions	F-22
F.4.1.6	System Service Calls	F-23
F.4.1.7	General Exceptions	F-23
F.4.1.7.1	Arithmetic Exceptions	F-23
F.4.1.7.2	Unaligned Access Exceptions	F-25
F.4.1.7.3	Illegal Instruction Exceptions	F-25
F.4.1.7.4	Invalid (Non-Canonical Virtual) Address Exceptions	F-26
F.4.1.7.5	Software Exceptions	F-26
F.4.1.7.6	Breakpoints and Debugger Support	F-27
F.4.1.7.7	Subsetted IEEE Instruction Exceptions	F-27
F.4.1.7.8	General Exceptions: Common Operations	F-28
F.4.1.8	Panic Exceptions	F-28
F.4.1.8.1	Kernel Stack Corruption	F-29
F.4.1.8.2	Unexpected Exceptions	F-29
F.4.1.8.3	Panic Exception Trap Frame and Dispatch	F-29
F.4.2	Interrupts	F-30
F.4.2.1	Interrupt Level Table (ILT)	F-30
F.4.2.2	Interrupt Mask Table (IMT)	F-30
F.4.2.3	Interrupt Dispatch Table (IDT)	F-31
F.4.2.4	Interrupt Dispatch	F-31

F.4.2.5	Interrupt Acknowledge	F-33
F.4.2.6	Synchronization Functions	F-33
F.4.2.7	Software Interrupt Requests	F-33
F.4.3	Machine Checks	F-34
F.4.3.1	Correctable Errors	F-35
F.4.3.2	Uncorrectable Errors	F-35
F.4.3.3	Machine Check Error Handling	F-35
F.4.3.4	Catastrophic Errors	F-37
F.5	.PALcode Instruction Descriptions	F-37
F.5.1	Privileged PALcode Instructions	F-38
F.5.1.1	Clear Software Interrupt Request	F-40
F.5.1.2	Disable Alignment Fixups	F-41
F.5.1.3	Disable All Interrupts	F-42
F.5.1.4	Drain All Aborts Including Machine Checks	F-43
F.5.1.5	Data Translation Buffer Invalidate Single	F-44
F.5.1.6	Enable Alignment Fixups	F-45
F.5.1.7	Enable Interrupts	F-46
F.5.1.8	Halt the Operating System by Trapping to Illegal Instruction	F-47
F.5.1.9	Initialize PALcode Data Structures with Operating System Values	F-48
F.5.1.10	Initialize Processor Control Region Data	F-50
F.5.1.11	Read the Software Event Counters	F-51
F.5.1.12	Read the Current IRQL from the PSR	F-52
F.5.1.13	Read Initial Kernel Stack Pointer for the Current Thread	F-53
F.5.1.14	Read the Machine Check Error Summary Register	F-54
F.5.1.15	Read the Processor Control Region Base Address	F-55
F.5.1.16	Read the Current Processor Status Register (PSR)	F-56
F.5.1.17	Read the Current Internal Processor State	F-57
F.5.1.18	Read the Thread Value for the Current Thread	F-58
F.5.1.19	Reboot — Transfer to Console Firmware	F-59
F.5.1.20	Restart the Operating System from the Restart Block	F-60
F.5.1.21	Return from System Service Call Exception	F-61
F.5.1.22	Return from Exception or Interrupt	F-63
F.5.1.23	Set Software Interrupt Request	F-65
F.5.1.24	Swap Thread Context	F-66
F.5.1.25	Swap the Current IRQL (Interrupt Request Level)	F-68
F.5.1.26	Swap the Initial Kernel Stack Pointer (IKSP) for the Current Thread	F-69
F.5.1.27	Swap the Currently Executing PALcode	F-70
F.5.1.28	Swap Process Context (Swap Address Space)	F-71
F.5.1.29	Translation Buffer Invalidate All	F-72
F.5.1.30	Translation Buffer Invalidate Multiple	F-73
F.5.1.31	Translation Buffer Invalidate Multiple for ASN	F-74
F.5.1.32	Translation Buffer Invalidate Single	F-75
F.5.1.33	Translation Buffer Invalidate Single for ASN	F-76
F.5.1.34	Write Kernel Exception Entry Routine	F-77
F.5.1.35	Write the Machine Check Error Summary Register	F-79
F.5.1.36	Write Performance Counter Interrupt Control Information	F-80
F.5.2	Unprivileged PALcode Instructions	F-81
F.5.2.1	Breakpoint Trap (Standard User-Mode Breakpoint)	F-82
F.5.2.2	Call Kernel Debugger	F-83
F.5.2.3	System Service Call	F-84
F.5.2.4	Generate a Trap	F-86
F.5.2.5	Instruction Memory Barrier	F-87
F.5.2.6	Kernel Breakpoint Trap	F-88
F.5.2.7	Read Thread Environment Block Pointer	F-89
F.5.3	Debug PALcode and Free PALcode	F-89
F.5.3.1	Kernel Stack Checking	F-89
F.5.3.2	Event Counters	F-90
F.6	Initialization and Firmware Transitions	F-90
F.6.1	Initialization	F-90

F.6.1.1	Pre-PALcode Initialization	F-91
F.6.1.2	PALcode Initialization	F-91
F.6.1.3	Kernel Callback Initialization of PALcode	F-92
F.6.1.4	Interrupt Table Initialization	F-92
F.6.2	Firmware Interfaces	F-92
F.6.2.1	Reboot Instruction – Transition to Firmware PALcode Context	F-92
F.6.2.2	Reboot and Restart Tasks and Sequence	F-94
F.6.2.3	Swppal Instruction – Transition to Any PALcode Environment	F-95
F.7	Windows NT Alpha Instruction Summary	F-96

Index

Instruction Index

Figures

1-1	Instruction Format Overview	1-4 (I)
2-1	Byte Format	2-1 (I)
2-2	Word Format	2-2 (I)
2-3	Longword Format	2-2 (I)
2-4	Quadword Format	2-2 (I)
2-5	F_floating Datum	2-3 (I)
2-6	F_floating Register Format	2-3 (I)
2-7	G_floating Datum	2-4 (I)
2-8	G_floating Register Format	2-4 (I)
2-9	D_floating Datum	2-5 (I)
2-10	D_floating Register Format	2-5 (I)
2-11	S_floating Datum	2-6 (I)
2-12	S_floating Register Format	2-7 (I)
2-13	T_floating Datum	2-8 (I)
2-14	T_floating Register Format	2-8 (I)
2-15	X_floating Datum	2-9 (I)
2-16	X_floating Register Format	2-9 (I)
2-17	X_floating Big-Endian Datum	2-10 (I)
2-18	X_floating Big-Endian Register Format	2-10 (I)
2-19	Longword Integer Datum	2-10 (I)
2-20	Longword Integer Floating-Register Format	2-10 (I)
2-21	Quadword Integer Datum	2-11 (I)
2-22	Quadword Integer Floating-Register Format	2-11 (I)
2-23	Little-Endian Byte Addressing	2-12 (I)
2-24	Big-Endian Byte Addressing	2-12 (I)
3-1	Memory Instruction Format	3-10 (I)
3-2	Memory Instruction with Function Code Format	3-10 (I)
3-3	Branch Instruction Format	3-11 (I)
3-4	Operate Instruction Format	3-11 (I)
3-5	Floating-Point Operate Instruction Format	3-12 (I)
3-6	PALcode Instruction Format	3-13 (I)
4-1	Floating-Point Control Register (FPCR) Format	4-81 (I)
4-2	Floating-Point Instruction Function Field	4-85 (I)
8-1	Alpha System Overview	8-1 (I)
10-1	Empty Absolute Longword Queue	10-21 (II-A)
10-2	Absolute Longword Queue with One Entry	10-21 (II-A)
10-3	Absolute Longword Queue with Two Entries	10-21 (II-A)
10-4	Absolute Longword Queue with Three Entries	10-22 (II-A)
10-5	Absolute Longword Queue with Three Entries After Removing the Second Entry	10-22 (II-A)
10-6	Empty Self-Relative Longword Queue	10-22 (II-A)
10-7	Self-Relative Longword Queue with One Entry	10-22 (II-A)
10-8	Self-Relative Longword Queue with Two Entries	10-23 (II-A)
10-9	Self-Relative Longword Queue with Three Entries	10-23 (II-A)
10-10	Empty Absolute Quadword Queue	10-24 (II-A)
10-11	Absolute Quadword Queue with One Entry	10-25 (II-A)
10-12	Absolute Quadword Queue with Two Entries	10-25 (II-A)
10-13	Absolute Quadword Queue with Three Entries	10-25 (II-A)
10-14	Absolute Quadword Queue with Three Entries After Removing the Second Entry	10-26 (II-A)
10-15	Empty Self-Relative Quadword Queue	10-26 (II-A)
10-16	Absolute Quadword Queue with One Entry	10-26 (II-A)
10-17	Self-Relative Quadword Queue with Two Entries	10-26 (II-A)
10-18	Self-Relative Quadword Queue with Three Entries	10-27 (II-A)
11-1	Virtual Address Format	11-2 (II-A)
11-2	Page Table Entry	11-3 (II-A)
12-1	Hardware Privileged Context Block	12-2 (II-A)
13-1	Address Space Number (ASN) Register	13-4 (II-A)

13-2	AST Enable (ASTEN) Register	13-5 (II-A)
13-3	AST Summary Register (ASTSR)	13-7 (II-A)
13-4	Data Alignment Trap Fixup (DATFX)	13-9 (II-A)
13-5	Executive Stack Pointer (ESP)	13-10 (II-A)
13-6	Floating Enable (FEN) Register	13-11 (II-A)
13-7	Interprocessor Interrupt Request (IPIR) Register	13-12 (II-A)
13-8	Interrupt Priority Level (IPL)	13-13 (II-A)
13-9	Machine Check Error Summary (MCES) Register	13-14 (II-A)
13-10	Performance Monitoring (PERFMON) Register	13-15 (II-A)
13-11	Privileged Context Block Base (PCBB) Register	13-16 (II-A)
13-12	Processor Base Register (PRBR)	13-17 (II-A)
13-13	Page Table Base Register (PTBR)	13-18 (II-A)
13-14	System Control Block Base (SCBB) Register	13-19 (II-A)
13-15	Software Interrupt Request Register (SIRR)	13-20 (II-A)
13-16	Software Interrupt Summary Register (SISR)	13-21 (II-A)
13-17	Supervisor Stack Pointer (SSP)	13-22 (II-A)
13-18	System Page Table Base Register (SYSPTBR)	13-23 (II-A)
13-19	Translation Buffer Check Register (TBCHK)	13-24 (II-A)
13-20	Translation Buffer Invalidate All (TBIA) Register	13-25 (II-A)
13-21	Translation Buffer Invalidate All Process (TBIAP) Register	13-26 (II-A)
13-22	Translation Buffer Invalidate Single (TBIS)	13-27 (II-A)
13-23	User Stack Pointer (USP)	13-28 (II-A)
13-24	Virtual Address Boundary (VIRBND) Register	13-29 (II-A)
13-25	Virtual Page Table Base (VPTB) Register	13-30 (II-A)
13-26	Who-Am-I (WHAMI) Register	13-31 (II-A)
14-1	Current Processor Status (PS Register)	14-5 (II-A)
14-2	Saved Processor Status (PS on Stack)	14-5 (II-A)
14-3	Program Counter (PC)	14-6 (II-A)
14-4	Exception Summary	14-12 (II-A)
14-5	Corrected Error and Machine Check Logout Frame	14-23 (II-A)
17-1	Virtual Address Format	17-2 (II-B)
17-2	Kseg Virtual Address Format	17-3 (II-B)
17-3	Page Table Entry (PTE)	17-4 (II-B)
17-4	Three-Level Page Table Mapping	17-10 (II-B)
18-1	Process Control Block (PCB)	18-2 (II-B)
19-1	Stack Frame Layout for callsys and rti	19-3 (II-B)
19-2	Stack Frame Layout for urti	19-3 (II-B)
19-3	Exception Summary Register	19-5 (II-B)
19-4	Machine Check Error Status (MCES) Register	19-8 (II-B)
22-1	Virtual Address Format	22-2 (II-C)
22-2	Kseg Virtual Address Format	22-3 (II-C)
22-3	Page Table Entry (PTE)	22-4 (II-C)
22-4	Three-Level Page Table Mapping	22-10 (II-C)
23-1	Process Control Block (PCB)	23-2 (II-C)
24-1	Stack Frame Layout for callsys and rti	24-3 (II-C)
24-2	Stack Frame Layout for urti	24-3 (II-C)
24-3	Exception Summary Register	24-5 (II-C)
24-4	Machine Check Error Status (MCES) Register	24-8 (II-C)
26-1	HWRPB Overview	26-2 (III)
26-2	Hardware Restart Parameter Block Structure	26-4 (III)
26-3	Per-CPU Slot in HWRPB	26-16 (III)
26-4	BIOS Emulator Register Structures	26-67 (III)
26-5	Console Data Structure Linkage	26-69 (III)
26-6	Console Routine Block	26-70 (III)
26-7	Console Terminal Block	26-74 (III)
26-8	CTB Fields	26-74 (III)
26-9	RXRDY and TXRDY Bitmasks in the HWRPB	26-76 (III)
26-10	Inter-Console Communications Buffer	26-77 (III)
27-1	Major State Transitions	27-2 (III)

27-2	Memory Data Descriptor (MEMDSC) Table	27-11 (III)
27-3	Static Memory Cluster Descriptor	27-11 (III)
27-4	MEMDSC Table with Null Memory Cluster Descriptor	27-13 (III)
27-5	Distributed Memory Cluster Descriptor	27-15 (III)
27-6	Distributed Memory Cluster Descriptors	27-17 (III)
27-7	Initial Virtual Memory Regions	27-19 (III)
27-8	Initial Page Tables	27-20 (III)
27-9	Alpha Disk Boot Block	27-40 (III)
27-10	Alpha ROM Boot Block	27-44 (III)
A-1	Branch-Format BSR and BR Opcodes	A-4
A-2	Memory-Format JSR Instruction	A-4
A-3	Bad Allocation in Cache	A-8
A-4	Better Allocation in Cache	A-8
A-5	Best Allocation in Cache	A-8
B-1	IEEE Floating-Point Control (FP_C) Quadword	B-4
B-2	IEEE Trap Handling Behavior	B-6
F-1	Processor Status Register	F-5
F-2	Virtual Address (Virtual View)	F-14
F-3	Virtual Address (Physical View)	F-15
F-4	Page Table Entry	F-16
F-5	Floating-Point Register Mask (FLOAT_REGISTER_MASK)	F-24
F-6	Integer Register Mask (INTEGER_REGISTER_MASK)	F-24
F-7	Software Interrupt Request Register	F-34
F-8	Machine Check Error Summary	F-36
F-9	PAL_BASE Internal Processor Register	F-93

Tables

2-1	F_floating Load Exponent Mapping (MAP_F)	2-3 (I)
2-2	S_floating Load Exponent Mapping (MAP_S)	2-7 (I)
3-1	Operand Notation	3-4 (I)
3-2	Operand Value Notation	3-4 (I)
3-3	Expression Operand Notation	3-4 (I)
3-4	Operand Name Notation	3-5 (I)
3-5	Operand Access Type Notation	3-5 (I)
3-6	Operand Data Type Notation	3-6 (I)
3-7	Operators	3-6 (I)
4-1	Opcod Qualifiers	4-3 (I)
4-2	Memory Integer Load/Store Instructions	4-4 (I)
4-3	Control Instructions Summary	4-19 (I)
4-4	Jump Instructions Branch Prediction	4-24 (I)
4-5	Integer Arithmetic Instructions Summary	4-25 (I)
4-6	Logical and Shift Instructions Summary	4-42 (I)
4-7	Byte-Within-Register Manipulation Instructions Summary	4-48 (I)
4-8	VAX Trapping Modes Summary	4-71 (I)
4-9	Summary of IEEE Trapping Modes	4-73 (I)
4-10	Trap Shadow Length Rules	4-76 (I)
4-11	Floating-Point Control Register (FPCR) Bit Descriptions	4-81 (I)
4-12	IEEE Floating-Point Function Field Bit Summary	4-85 (I)
4-13	VAX Floating-Point Function Field Bit Summary	4-87 (I)
4-14	Memory Format Floating-Point Instructions Summary	4-90 (I)
4-15	Floating-Point Branch Instructions Summary	4-99 (I)
4-16	Floating-Point Operate Instructions Summary	4-101 (I)
4-17	Miscellaneous Instructions Summary	4-132 (I)
4-18	VAX Compatibility Instructions Summary	4-152 (I)
5-1	Processor Issue Constraints	5-13 (I)
6-1	PALcode Instructions that Require Recognition	6-4 (I)
6-2	Required PALcode Instructions	6-4 (I)
10-1	OpenVMS PALcode Instructions	10-1 (II-A)
10-2	Unprivileged General PALcode Instruction Summary	10-3 (II-A)
10-3	Queue PALcode Instruction Summary	10-28 (II-A)
10-4	Unprivileged PALcode Thread Instructions	10-78 (II-A)
10-5	PALcode Privileged Instructions Summary	10-81 (II-A)
11-1	Virtual Address Options	11-3 (II-A)
11-2	Page Table Entry	11-4 (II-A)
13-1	Internal Processor Register (IPR) Summary	13-2 (II-A)
13-2	Internal Processor Register (IPR) Access Summary	13-3 (II-A)
14-1	Exceptions, Interrupts, and Machine Checks Summary	14-3 (II-A)
14-2	Processor Status Register Summary	14-5 (II-A)
14-3	Stack Frame	14-8 (II-A)
14-4	Exception Summary	14-12 (II-A)
14-5	Corrected Error and Machine Check Logout Frame Fields	14-23 (II-A)
14-6	System Control Block Summary	14-24 (II-A)
14-7	SCB Entries for Faults	14-25 (II-A)
14-8	SCB Entries for Arithmetic Traps	14-25 (II-A)
14-9	SCB Entries for Asynchronous System Traps	14-26 (II-A)
14-10	SCB Entries for Data Alignment Trap	14-26 (II-A)
14-11	SCB Entries for Other Synchronous Traps	14-26 (II-A)
14-12	SCB Entries for Processor Software Interrupts	14-27 (II-A)
14-13	SCB Entries for Processor Hardware Interrupts and Machine Checks	14-28 (II-A)
14-14	Processor State Transitions	14-35 (II-A)
15-1	Tru64 UNIX Register Usage	15-1 (II-B)
15-2	Code Flow Constants and Terms	15-2 (II-B)
15-3	Machine State Terms	15-2 (II-B)

16-1	Unprivileged PALcode Instructions	16-1 (II-B)
16-2	Privileged PALcode Instructions	16-10 (II-B)
17-1	Virtual Address Space Segments	17-1 (II-B)
17-2	Virtual Address Options	17-2 (II-B)
17-3	Page Table Entry (PTE) Bit Summary	17-4 (II-B)
17-4	Memory-Management Fault Type Codes	17-13 (II-B)
19-1	Processor Status Summary	19-2 (II-B)
19-2	Entry Point Address Registers	19-4 (II-B)
19-3	Exception Summary Register Bit Definitions	19-5 (II-B)
19-4	System Entry Hardware Interrupts	19-7 (II-B)
19-5	Machine Check Error Status (MCES) Register Bit Definitions	19-8 (II-B)
20-1	Alpha Linux Register Usage	20-1 (II-C)
20-2	Code Flow Constants and Terms	20-2 (II-C)
20-3	Machine State Terms	20-2 (II-C)
21-1	Unprivileged PALcode Instructions	21-1 (II-C)
21-2	Privileged PALcode Instructions	21-9 (II-C)
22-1	Virtual Address Space Segments	22-1 (II-C)
22-2	Virtual Address Options	22-2 (II-C)
22-3	Page Table Entry (PTE) Bit Summary	22-4 (II-C)
22-4	Memory-Management Fault Type Codes	22-13 (II-C)
24-1	Processor Status Summary	24-2 (II-C)
24-2	Entry Point Address Registers	24-4 (II-C)
24-3	Exception Summary Register Bit Definitions	24-5 (II-C)
24-4	System Entry Hardware Interrupts	24-7 (II-C)
24-5	Machine Check Error Status (MCES) Register Bit Definitions	24-8 (II-C)
26-1	HWRPB Fields	26-6 (III)
26-2	System Variation Field (HWRPB[88])	26-12 (III)
26-3	Granularity Hint Fields	26-14 (III)
26-4	Per-CPU Slot Fields	26-16 (III)
26-5	Per-CPU State Flags	26-22 (III)
26-6	Required Environment Variables	26-26 (III)
26-7	Supported Languages	26-28 (III)
26-8	Supported Character Sets	26-29 (III)
26-9	Console Callback Routines	26-30 (III)
26-10	CRB Fields	26-70 (III)
26-11	Inter-Console Communications Buffer Fields	26-77 (III)
27-1	Effects of Power-Up Initialization	27-4 (III)
27-2	Tru64 UNIX and Alpha Linux PALcode Switching	27-7 (III)
27-3	Processor State at Exit from swppal	27-8 (III)
27-4	Memory Data Descriptor Table Fields	27-11 (III)
27-5	Static Memory Cluster Descriptor Fields	27-12 (III)
27-6	MEMDSC Table Fields with Null Memory Cluster Descriptor	27-13 (III)
27-7	Distributed Memory Cluster Descriptor Fields	27-15 (III)
27-8	Console Interpretation of BIP and RC Flags	27-22 (III)
27-9	Processor Initialization	27-23 (III)
27-10	Initial HWPCB Contents	27-25 (III)
27-11	Bootstrap Devices and Image Media	27-39 (III)
A-1	Prefetch Instruction Support	A-10
A-2	Decodable Pseudo-Operations (Stylized Code Forms)	A-16
B-1	Floating-Point Control (FP_C) Quadword Bit Summary	B-5
B-2	IEEE Floating-Point Trap Handling	B-7
B-3	IEEE Standard Charts	B-11
C-1	Instruction Format and Opcode Notation	C-1
C-2	Common Architecture Instructions	C-2
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-10
C-7	Key to Opcode Summary	C-11

C-8	Common Architecture Opcodes in Numerical Order	C-12
C-9	OpenVMS Unprivileged PALcode Instructions	C-16
C-10	OpenVMS Privileged PALcode Instructions	C-17
C-11	Tru64 UNIX Unprivileged PALcode Instructions	C-19
C-12	Tru64 UNIX Privileged PALcode Instructions	C-19
C-13	Alpha Linux Unprivileged PALcode Instructions	C-20
C-14	Alpha Linux Privileged PALcode Instructions	C-20
C-15	PALcode Opcodes in Numerical Order	C-21
C-16	Required PALcode Opcodes	C-24
C-17	Opcodes Reserved for PALcode	C-24
C-18	Opcodes Reserved for Compaq	C-25
C-19	ASCII Character Set	C-26
D-1	Processor Type Assignments	D-1
D-2	PALcode Variation Assignments	D-4
D-3	AMASK Bit Assignments	D-4
D-4	IMPLVER Value Assignments	D-5
E-1	21064/21066/21068 Performance Monitoring Functions	E-9
E-2	21064/21066/21068 MUX Control Fields in ICCSR Register	E-11
E-3	Performance Monitoring Functions	E-13
E-4	21164/21164PC Enable Counters	E-16
E-5	21164/21164PC Disable Counters	E-16
E-6	21164 Select Desired Events	E-16
E-7	21164PC Select Desired Events	E-16
E-8	21164/21164PC Select Special Options	E-17
E-9	21164/21164PC Select Desired Frequencies	E-18
E-10	21164/21164PC Read Counters	E-18
E-11	21164/21164PC Write Counters	E-19
E-12	21164/21164PC Counter 1 (PCSEL1) Event Selection	E-19
E-13	21164/21164PC Counter 2 (PCSEL2) Event Selection	E-20
E-14	21164 CBOX1 Event Selection	E-20
E-15	21164 CBOX2 Event Selection	E-21
E-16	21164PC PM0_MUX Event Selection	E-21
E-17	21164PC PM1_MUX Event Selection	E-21
E-18	Performance Monitoring Functions	E-23
E-19	21264 and 21364 Enable Counters	E-25
E-20	21264 and 21364 Disable Counters	E-25
E-21	21264 and 21364 Select Desired Events	E-26
E-22	21264 and 21364 Read Counters	E-27
E-23	21264 and 21364 Write Counters	E-28
E-24	21264 and 21364 Enable and Write Counters	E-28
E-25	21264 and 21364 Read I_STAT Values	E-28
E-26	21264 and 21364 Read PMPC Value	E-31
F-1	General-Purpose Integer Registers	F-4
F-2	General-Purpose Floating-Point Registers	F-4
F-3	Processor Status Register Fields	F-5
F-4	Processor Status Register IRQL Field Summary	F-5
F-5	Processor Privilege Mode Map	F-6
F-6	Internal Processor Register Summary	F-6
F-7	Internal Processor Registers	F-7
F-8	Virtual Address Map	F-13
F-9	I/O Address Extension Address Map	F-14
F-10	Page Table Entry Fields	F-16
F-11	Translation Buffer Management Instructions	F-17
F-12	Trap Frame Definitions	F-22
F-13	Exception Summary Register (EXCEPTION_SUMMARY)	F-25
F-14	Exception Summary Register Fields	F-25
F-15	Breakpoint Types	F-27
F-16	Interrupt Mask Table (IMT)	F-30
F-17	Software Entries of the IMT	F-31

F-18	Software Interrupt Request Register Fields	F-34
F-19	Machine Check Error Summary Fields	F-36
F-20	Machine Check Types	F-36
F-21	Privileged PALcode Instruction Summary	F-38
F-22	Exception Class Values	F-78
F-23	Unprivileged PALcode Instruction Summary	F-81
F-24	Windows NT Alpha Unprivileged PALcode Instructions	F-96
F-25	Windows NT Alpha Privileged PALcode Instructions	F-96

Preface

The *Alpha Architecture Reference Manual* is organized as shown in the following table.

Name	Symbol	Contents
Common Architecture		
(I)		Describes the architecture that is common to and required of all implementations, and contains the following chapters: <ul style="list-style-type: none">• Chapter 1, Introduction to the Common Architecture (I)• Chapter 2, Basic Architecture (I)• Chapter 3, Instruction Formats (I)• Chapter 4, Instruction Descriptions (I)• Chapter 5, System Architecture and Programming Implications (I)• Chapter 6, Common PALcode Architecture (I)• Chapter 7, Console Subsystem Overview (I)• Chapter 8, Input/Output Overview (I)
OpenVMS Operating System PALcode Architecture		
(II-A)		Describes how the OpenVMS operating system relates to the Alpha architecture and contains the following chapters: <ul style="list-style-type: none">• Chapter 9, Introduction to OpenVMS (II-A)• Chapter 10, PALcode Instruction Descriptions (II-A)• Chapter 11, Memory Management (II-A)• Chapter 12, Process Structure (II-A)• Chapter 13, Internal Processor Registers (II-A)• Chapter 14, Exceptions, Interrupts, and Machine Checks (II-A)
Tru64 UNIX Operating System PALcode Architecture		
(II-B)		Describes how the Tru64 UNIX operating system relates to the Alpha architecture and contains the following chapters: <ul style="list-style-type: none">• Chapter 15, Introduction to Tru64 UNIX (II-B)• Chapter 16, PALcode Instruction Descriptions (II-B)• Chapter 17, Memory Management (II-B)• Chapter 18, Process Structure (II-B)• Chapter 19, Exceptions and Interrupts (II-B)

Alpha Linux Operating System PALcode Architecture

(II-C) Describes how the Alpha Linux operating system relates to the Alpha architecture and contains the following chapters:

- Chapter 20, Introduction to Alpha Linux (II-C)
- Chapter 21, PALcode Instruction Descriptions (II-C)
- Chapter 22, Memory Management (II-C)
- Chapter 23, Process Structure (II-C)
- Chapter 24, Exceptions and Interrupts (II-C)

Console Interface Architecture

(III) Describes an architected console firmware implementation and contains the following chapters:

- Chapter 25, Console Subsystem Overview (III)
- Chapter 26, Console Interface to Operating System Software (III)
- Chapter 27, System Bootstrapping (III)

Appendixes

The following appendixes are included:

- Appendix A, Software Considerations
- Appendix B, IEEE Floating-Point Conformance
- Appendix C, Instruction Summary
- Appendix D, Registered System and Processor Identifiers
- Appendix E, Waivers and Implementation-Dependent Functionality
- Appendix F, Windows NT Software

Indexes

The index at the end of the manual is structured like a master index. Index entries are called out by the chapter and page, followed by the appropriate Section symbol: (I), (II-A), and so forth. Index entries for the appendixes are called out by appendix letter and page number. Following the manual index is an index of the instructions. The instruction index is the easiest way to find primary documentation for the Alpha instruction set and the PALcode instructions for each operating system.

Common Architecture (I)

The following chapters describe the common Alpha architecture:

- Chapter 1, Introduction to the Common Architecture (I)
- Chapter 2, Basic Architecture (I)
- Chapter 3, Instruction Formats (I)
- Chapter 4, Instruction Descriptions (I)
- Chapter 5, System Architecture and Programming Implications (I)
- Chapter 6, Common PALcode Architecture (I)
- Chapter 7, Console Subsystem Overview (I)
- Chapter 8, Input/Output Overview (I)

Introduction to the Common Architecture (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, Tru64 UNIX, and Alpha Linux 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.

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 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, Tru64 UNIX, and Alpha Linux operating systems. PALcode can provide this functionality with little overhead. For example, the OpenVMS PALcode instructions let Alpha run OpenVMS with little more hardware than that found on a conventional RISC machine: the PALmode bit itself, plus four 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

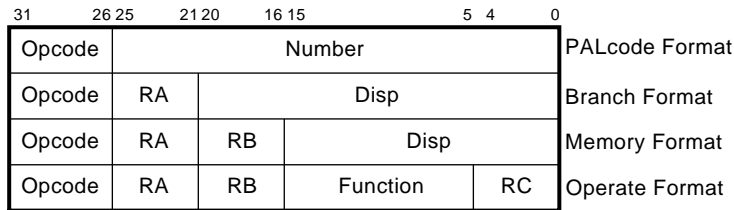
Alpha is a load/store RISC architecture with the following data characteristics:

- 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. 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



- **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 in Section 1.1, 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, and they can 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 operations 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
- A set of multimedia instructions that support graphics and video

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.
- 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 integer register file to a floating-point register file, and data in a 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 moves 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, and IEEE extended, double, and 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 (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 that 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 two periods 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 Illustration Conventions

Illustrations 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 Chapter 10, or are stylized code forms found in Appendix A .

Basic Architecture (I)

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.

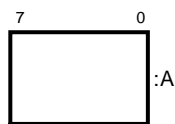
2.2 Data Types

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

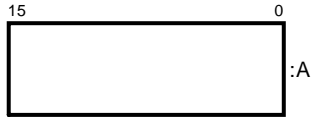


A byte is specified by its address A. A byte is an 8-bit value. The byte is only supported in Alpha by the load, store, sign-extend, extract, mask, insert, and zap instructions.

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



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



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.

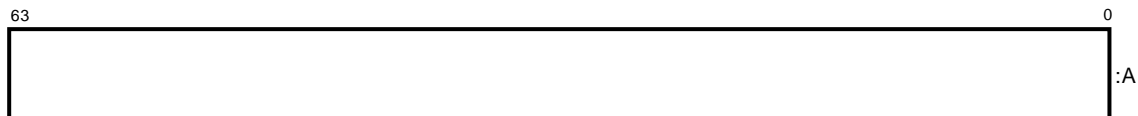
Note:

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.

Figure 2–4 Quadword Format



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.)

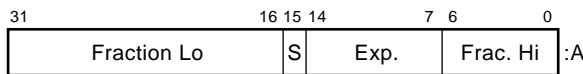
2.2.5 VAX Floating-Point Formats

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

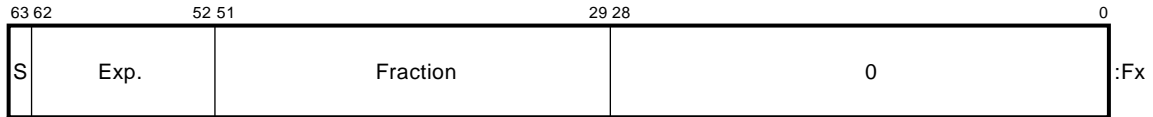
An F_floating datum is 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2-5.

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. This mapping preserves both normal values and exceptional values.

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

Memory <14:7>	Register <62:52>
1 1111111	1 000 1111111
1 xxxxxxx	1 000 xxxxxxx (xxxxxxx not all 1's)
0 xxxxxxx	0 111 xxxxxxx (xxxxxxx not all 0's)
0 0000000	0 000 0000000

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.

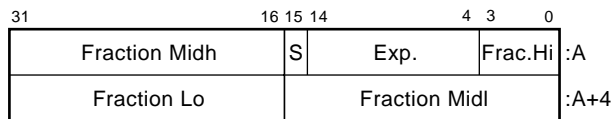
Note:

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

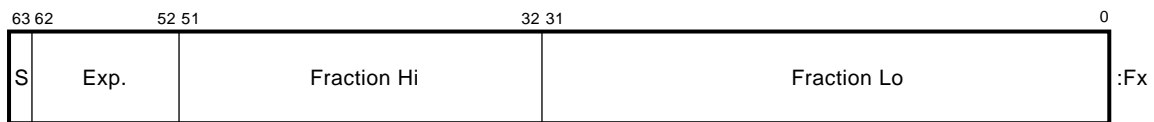
A G_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2-7.

Figure 2-7: G_floating Datum



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

Figure 2-8 G_floating Register Format



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.

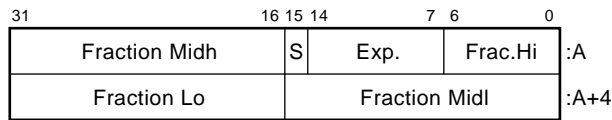
Note:

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

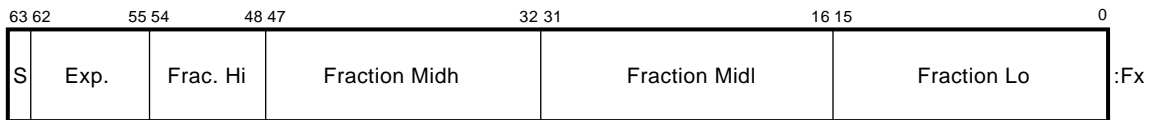
A D_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2-9.

Figure 2-9: D_floating Datum



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



The reordering of bits required for a D_floating load or store is identical to that 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 three 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:

- P – the number of fraction bits
- Emax – the maximum exponent
- Emin – the minimum exponent

Within each format, only the following entities are permitted:

- Numbers of the form $(-1)^S \times 2^E \times b(0).b(1)b(2)..b(P-1)$ where:
 - S = 0 or 1
 - E = any integer between Emin and Emax, inclusive
 - b(n) = 0 or 1
- Two infinities – positive and negative
- At least one Signaling NaN
- At least one Quiet NaN

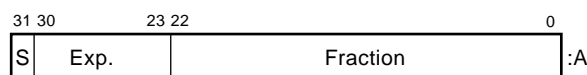
NaN is an acronym for Not-a-Number. A NaN is an IEEE floating-point bit pattern that represents something other than a number. NaNs come in two forms: Signaling NaNs and Quiet NaNs. Signaling NaNs are used to provide values for uninitialized variables and for arithmetic enhancements. Quiet NaNs provide 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

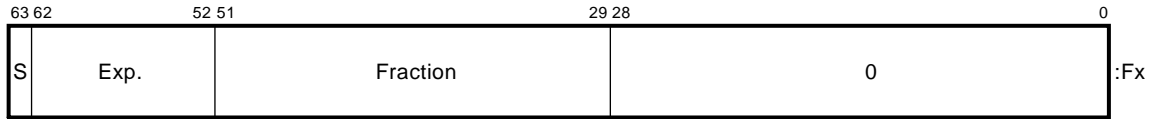
An IEEE single-precision, or S_floating, datum occupies 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2–11.

Figure 2–11: S_floating Datum



An S_floating operand occupies 64 bits in a floating register, left-justified in the 64-bit register, as shown in Figure 2–12.

Figure 2–12 S_floating Register Format



The S_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_floating number, suitable for either S_floating or T_floating operations. The mapping from 8-bit memory-format exponents to 11-bit register-format exponents is shown in Table 2–2.

Table 2–2: S_floating Load Exponent Mapping (MAP_S)

Memory <30:23>	Register <62:52>
1 1111111	1 111 1111111
1 xxxxxxx	1 000 xxxxxxx (xxxxxxx not all 1's)
0 xxxxxxx	0 111 xxxxxxx (xxxxxxx not all 0's)
0 0000000	0 000 0000000

This mapping preserves both normal values and exceptional values. Note that the mapping for all 1's differs from that of F_floating load, since for S_floating all 1's is an exceptional value and for F_floating all 1's is a normal value.

The S_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_floating load instruction does no checking of the input.

The S_floating store instruction does no checking of the data; the preceding operation should have specified an S_floating result.

An S_floating datum is specified by its address A, the address of the byte containing bit 0. The memory form of an S_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:

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

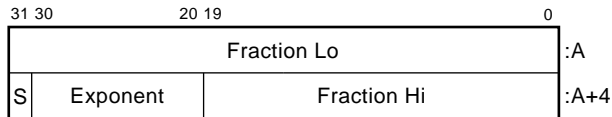
Note:

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

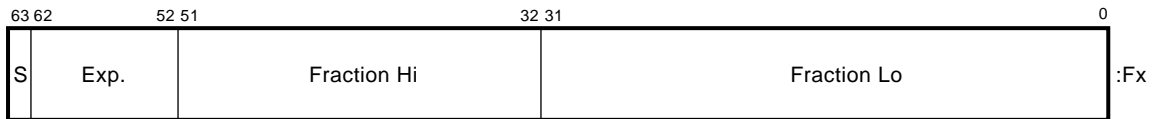
An IEEE double-precision, or T_floating, datum occupies 8 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–13.

Figure 2–13: T_floating Datum



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



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:

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

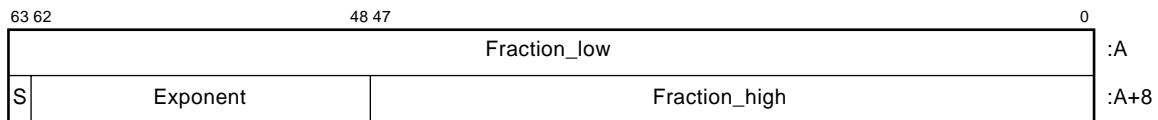
Alpha implementations will impose a significant performance penalty when accessing T_floating operands that are not naturally aligned. (A naturally aligned T_floating datum has zero as the low-order three bits of its address.)

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.

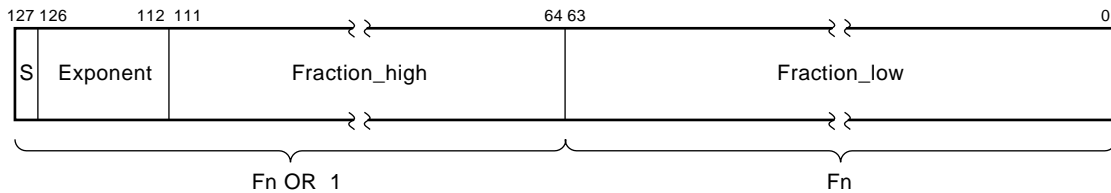
An IEEE extended-precision, or X_floating, datum occupies 16 contiguous bytes in memory, starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 127, as shown in Figure 2–15.

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.

Figure 2–16: X_floating Register Format



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:

- If $E=32767$ and $F \neq 0$, then V is a NaN, regardless of S.
- If $E=32767$ and $F=0$, then $V = (-1)^{S} \times \text{Infinity}$.
- If $0 < E < 32767$, then $V = (-1)^{S} \times 2^{(E-16383)} \times (1.F)$.
- If $E=0$ and $F \neq 0$, then $V = (-1)^{S} \times 2^{(-16382)} \times (0.F)$.
- If $E = 0$ and $F = 0$, then $V = (-1)^{S} \times 0$ (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.

Figure 2–17 X_floating Big-Endian Datum

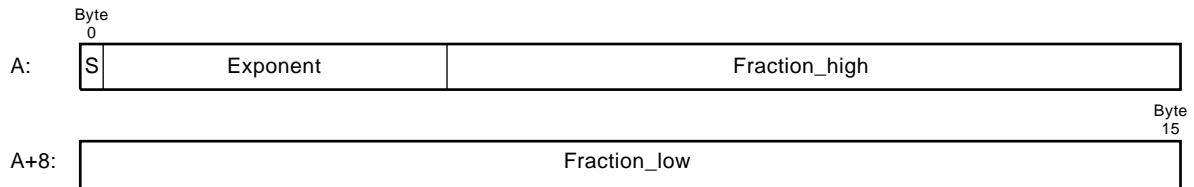
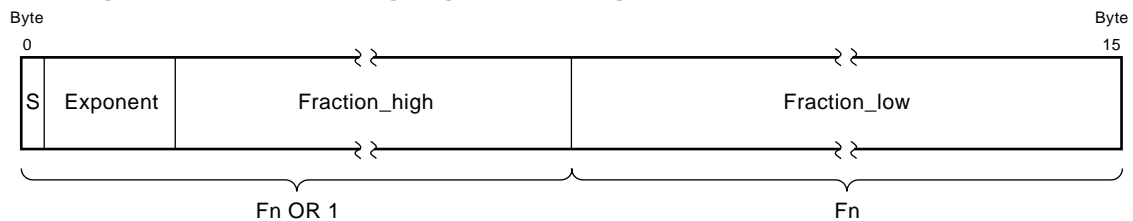


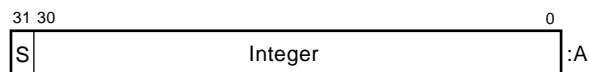
Figure 2–18: X_floating Big-Endian Register Format



2.2.7 Longword Integer Format in Floating-Point Unit

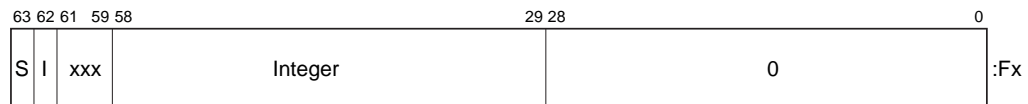
A longword integer operand occupies 32 bits in memory, arranged as shown in Figure 2–19.

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.

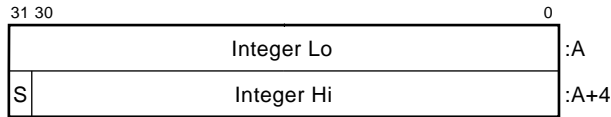
Note:

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

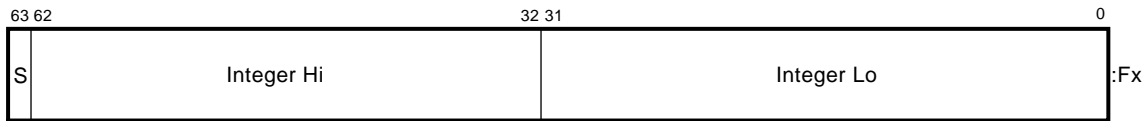
A quadword integer operand occupies 64 bits in memory, arranged as shown in Figure 2–21.

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 between memory and the floating registers. (The ITOFT and FTOIT are used to move quadword data between integer and 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

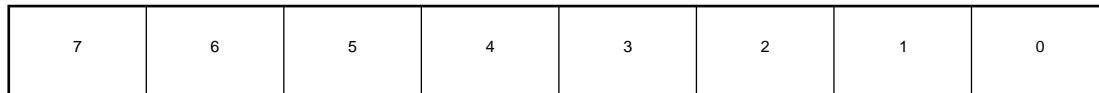
- Leading Separate Numeric String
- Packed Decimal String

2.3 Big-Endian Addressing Support

Alpha implementations may include optional big-endian addressing support.

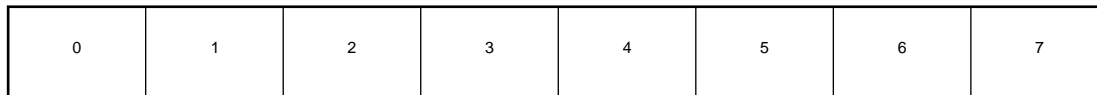
In a little-endian machine, the bytes within a quadword are numbered right to left:

Figure 2–23 Little-Endian Byte Addressing



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

Figure 2–24 Big-Endian Byte Addressing



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

The format for the `X_floating 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.
- Manipulation of 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 $va\langle 2 \rangle$ (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 word-length load and store instructions, LDWU and STW, invert bits $va\langle 1:2 \rangle$ (bits 1 and 2 of the virtual address). This has the effect of accessing the half of the longword that would be accessed under the little-endian convention.
- If the big-endian convention is chosen, the byte-length load and store instructions, LDBU and STB, invert bits $va\langle 0:2 \rangle$ (bits 0 through 2 of the virtual address). This has the effect of accessing the half of the word that would be accessed under the little-endian convention.
- If the big-endian convention is chosen, the byte manipulation instructions (EXT $_{xx}$, INS $_{xx}$, MSK $_{xx}$) invert bits $Rbv\langle 2:0 \rangle$. 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 $0 \text{ MOD } 8$. The same bytes accessed by a longword load/store instruction have data-stream address $0 \text{ MOD } 8$ under the little-endian convention, and $4 \text{ MOD } 8$ 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 Section A.4.3 for a suggested code sequence.

Instruction Formats (I)

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 long-word-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.

Implementation note:

As described in Appendix A, certain load instructions to an R31 destination are the preferred method for performing a cache block prefetch.

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(dis)}` 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.

A memory management exception or alignment exception is never signaled for a load that specifies F31 as a destination register. It is UNPREDICTABLE whether a floating-point disabled exception can be signaled by a load that specifies F31 as a destination register. For all other instructions that specify F31 as an output operand, 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.

Implementation note:

As described in Appendix A, certain load instructions to an F31 destination are the preferred method for signalling a cache block prefetch.

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, Tru64 UNIX, and Alpha Linux supply a per-thread 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 4.11.9.

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

Notation	Meaning
Ra	An integer register operand in the Ra field of the instruction
Rb	An integer register operand in the Rb field of the instruction
#b	An integer literal operand in the Rb field of the instruction
Rc	An integer register operand in the Rc field of the instruction
Fa	A floating-point register operand in the Ra field of the instruction
Fb	A floating-point register operand in the Rb field of the instruction
Fc	A floating-point register operand in the Rc field of the instruction

Table 3–2 Operand Value Notation

Notation	Meaning
Rav	The value of the Ra operand. This is the contents of register Ra.
Rbv	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.
Fav	The value of the floating-point Fa operand. This is the contents of register Fa.
Fbv	The value of the floating-point Fb operand. This is the contents of register Fb.

Table 3–3 Expression Operand Notation

Notation	Meaning
IPR_x	Contents of Internal Processor Register x
IPR_SP[mode]	Contents of the per-mode stack pointer selected by mode
PC	Updated PC value
Rn	Contents of integer register n
Fn	Contents of floating-point register n
X[m]	Element m of array X

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>

3.2.2.1 Operand Name Notation

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 3–4 Operand Name Notation

Name	Meaning
disp	The displacement field of the instruction
fnc	The PALcode function field of the instruction
Ra	An integer register operand in the Ra field of the instruction
Rb	An integer register operand in the Rb field of the instruction
#b	An integer literal operand in the Rb field of the instruction
Rc	An integer register operand in the Rc field of the instruction
Fa	A floating-point register operand in the Ra field of the instruction
Fb	A floating-point register operand in the Rb field of the instruction
Fc	A floating-point register operand in the Rc field of the instruction

3.2.2.2 Operand Access Type Notation

A letter that denotes the operand access type:

Table 3–5 Operand Access Type Notation

Access Type	Meaning
a	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).
i	The operand is an immediate literal in the instruction.
r	The operand is read only.
m	The operand is both read and written.
w	The operand is write only.

3.2.2.3 Operand Data Type Notation

A letter that denotes the data type of the operand:

Table 3–6 Operand Data Type Notation

Data Type	Meaning
b	Byte
f	F_floating
g	G_floating
l	Longword
q	Quadword
s	IEEE single floating (S_floating)
t	IEEE double floating (T_floating)
w	Word
x	The data type is specified by the instruction

3.2.3 Operators

Table 3–7 describes the operators:

Table 3–7 Operators

Operator	Meaning
!	Comment delimiter.
+	Addition.
-	Subtraction.
*	Signed multiplication.
*U	Unsigned multiplication.
**	Exponentiation (left argument raised to right argument).
/	Division.
←	Replacement.
	Bit concatenation.
{ }	Indicates explicit operator precedence.
(x)	Contents of memory location whose address is x.
x <m:n>	Contents of bit field of x defined by bits n through m.
x <m>	M'th bit of x.
ACCESS(x,y)	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.

Table 3–7 Operators (Continued)

Operator	Meaning
AND	Logical product.
ARITH_RIGHT_SHIFT(x,y)	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.
BYTE_ZAP(x,y)	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 $y \langle n \rangle \leftrightarrow x \langle 8n+7:8n \rangle$. This correspondence also exists between y and the result. For each bit of y from $n = 0$ to 7, if $y \langle n \rangle$ is 0 then byte $\langle n \rangle$ of x is copied to byte $\langle n \rangle$ of result, and if $y \langle n \rangle$ is 1 then byte $\langle n \rangle$ of result is forced to all zeros.
CASE	The CASE construct selects one of several actions based on the value of its argument. The form of a case is: <pre> CASE argument OF argvalue1: action_1 argvalue2: action_2 ... argvaluen:action_n [otherwise: default_action] ENDCASE </pre> If the value of argument is argvalue1 then action_1 is executed; if argument = argvalue2, then action_2 is executed, and so forth. Once a single action is executed, the code stream breaks to the END-CASE (there is an implicit break as in Pascal). Each action may nonetheless be a sequence of pseudocode operations, one operation per line. 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.
DIV	Integer division (truncates).
LEFT_SHIFT(x,y)	Logical left shift of first operand by the second operand.Y is an unsigned shift value. Zeros are moved into the vacated bit positions, and shifted out bits are discarded.
LOAD_LOCKED	The processor records the target physical address in a per-processor locked_physical_address register and sets the per-processor lock_flag.
lg	Log to the base 2.
MAP_x	F_float or S_float memory-to-register exponent mapping function.
MAXS(x,y)	Returns the larger of x and y, with x and y interpreted as signed integers.

Table 3–7 Operators (Continued)

Operator	Meaning
MAXU(x,y)	Returns the larger of x and y, with x and y interpreted as unsigned integers.
MINS(x,y)	Returns the smaller of x and y, with x and y interpreted as signed integers.
MINU(x,y)	Returns the smaller of x and y, with x and y interpreted as unsigned integers.
x MOD y	x modulo y.
NOT	Logical (ones) complement.
OR	Logical sum.
PHYSICAL_ADDRESS	Translation of a virtual address.
PRIORITY_ENCODE	Returns the bit position of most significant set bit, interpreting its argument as a positive integer (=int(lg(x))). For example: <pre>priority_encode(255) = 7</pre>

Relational Operators:

Operator	Meaning
LT	Less than signed
LTU	Less than unsigned
LE	Less or equal signed
LEU	Less or equal unsigned
EQ	Equal signed and unsigned
NE	Not equal signed and unsigned
GE	Greater or equal signed
GEU	Greater or equal unsigned
GT	Greater signed
GTU	Greater unsigned
LBC	Low bit clear
LBS	Low bit signed

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.

Table 3–7 Operators (Continued)

Operator	Meaning
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:

- Only operands that appear on the left side of a replacement operator are modified.
- No operator precedence is assumed other than that replacement (\leftarrow) has the lowest precedence. Explicit precedence is indicated by the use of "{}".
- 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
- PALcode

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.

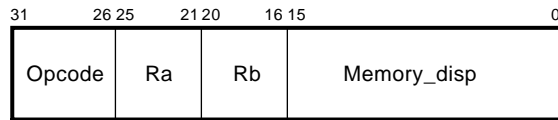
Software Note:

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.

Figure 3–1: Memory Instruction Format



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):

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

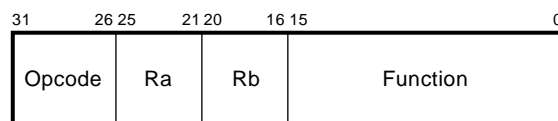
For LDAH the virtual address (va) is computed as follows:

$$va \leftarrow \{Rbv + \text{SEXT}(\text{Memory_disp} * 65536)\}$$

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.

Figure 3–2: Memory Instruction with Function Code Format



The memory instruction with function code format contains a 6-bit opcode field and a 16-bit function field. Unused function codes produce UNPREDICTABLE but not UNDEFINED results; they are not security holes.

There are two fields, Ra and Rb. The usage of those fields depends on the instruction. See Section 4.11.

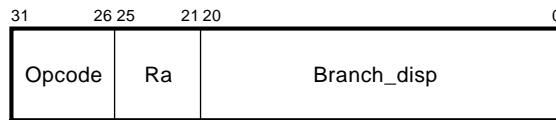
3.3.1.2 Memory Format Jump Instructions

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.

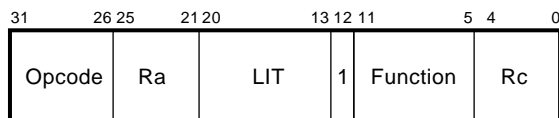
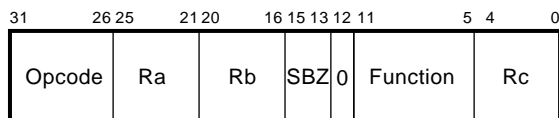
The displacement is treated as a longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits, and added to the updated PC to form the target virtual address. Overflow is ignored in this calculation. The target virtual address (va) is computed as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

$$va \leftarrow PC + \{4 * \text{SEXT}(\text{Branch_disp})\}$$

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 code field. Unused function codes for 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, 1C, 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:

```

IF inst<25:21> EQ 31 THEN
    Rav ← 0
ELSE
    Rav ← Ra
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:

```

IF inst <12> EQ 1 THEN
    Rbv ← ZEXT(inst<20:13>)
ELSE
    IF inst <20:16> EQ 31 THEN
        Rbv ← 0
    ELSE
        Rbv ← Rb
    END
END

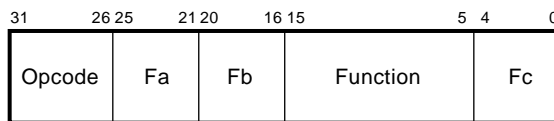
```

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



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, 1C, 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:

```
IF inst<25:21> EQ 31 THEN
    Fav ← 0
ELSE
    Fav ← Fa
END
```

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

```
IF inst<20:16> EQ 31 THEN
    Fbv ← 0
ELSE
    Fbv ← Fb
END
```

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.

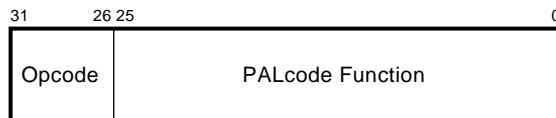
3.3.4.2 Floating-Point/Integer Register Moves

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



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.

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:

Instruction Type	Section
Integer load and store	4.2
Integer control	4.3
Integer arithmetic	4.4
Logical and shift	4.5
Byte manipulation	4.6
Floating-point load and store	4.7
Floating-point control	4.8
Floating-point branch	4.9
Floating-point operate	4.10
Miscellaneous	4.11
VAX compatibility	4.12
Multimedia (graphics and video)	4.13

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 PALcode. System software may provide emulation routines for subsetted instructions.

4.1.2 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 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 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.

4.1.3 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 lay-

ered 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.

4.1.4 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 4–1: Opcode Qualifiers

Qualifier	Meaning
C	Chopped rounding
D	Rounding mode dynamic
M	Round toward minus infinity
I	Inexact result enable
S	Exception completion enable
U	Floating underflow enable
V	Integer overflow enable

The default values are normal rounding, exception 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 4–2: Memory Integer Load/Store Instructions

Mnemonic	Operation
LDA	Load Address
LDAH	Load Address High
LDBU	Load Zero-Extended Byte from Memory to Register
LDL	Load Sign-Extended Longword
LDL_L	Load Sign-Extended Longword Locked
LDQ	Load Quadword
LDQ_L	Load Quadword Locked
LDQ_U	Load Quadword Unaligned
LDWU	Load Zero-Extended Word from Memory to Register
STB	Store Byte
STL	Store Longword
STL_C	Store Longword Conditional
STQ	Store Quadword
STQ_C	Store Quadword Conditional
STQ_U	Store Quadword Unaligned
STW	Store Word

4.2.1 Load Address

Format:

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

Operation:

$Ra \leftarrow Rbv + \text{SEXT}(\text{disp})$!LDA
 $Ra \leftarrow Rbv + \text{SEXT}(\text{disp} * 65536)$!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:

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

CASE

big_endian_data: va' \leftarrow va XOR 000₂ !LDQ

big_endian_data: va' \leftarrow va XOR 100₂ !LDL

big_endian_data: va' \leftarrow va XOR 110₂ !LDWU

big_endian_data: va' \leftarrow va XOR 111₂ !LDBU

little_endian_data: va' \leftarrow va

ENDCASE

Ra \leftarrow (va')<63:0> !LDQ

Ra \leftarrow SEXT((va')<31:0>) !LDL

Ra \leftarrow ZEXT((va')<15:0>) !LDWU

Ra \leftarrow 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.
- An LDL instruction for which the Ra operand is 31 is executed as a PREFETCH instruction, described in Section 4.11.8.
- An LDQ instruction for which the Ra operand is 31 is executed as a PREFETCH_EN instruction, described in Section 4.11.8.

Implementation Notes:

- The LDWU and LDBU instructions are supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 0. LDWU and LDBU are supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 0. Software emulation of LDWU and LDBU is significantly slower than hardware support.
- 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:

$$va \leftarrow \{ \{Rbv + \text{SEXT}(\text{disp})\} \text{ AND NOT } 7 \}$$
$$Ra \leftarrow (va) \langle 63:0 \rangle$$

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 0002    ! LDQ_L
  big_endian_data: va' ← va XOR 1002    ! 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' is computed from va by inverting va<2> (bit 2 of the virtual address), but any memory management fault is reported for the original 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 an 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_condition
- Executing a WH64x 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 WH64x or ECB instruction.

In addition, a set lock_flag on processor *B* can be unpredictably cleared by unspecified events on processor *A*. But, processor *A* will guarantee that such events are rare enough that they will not interfere with the forward progress of the system.

Implementation Note:

Processor *A* can, at the implementation's option, cause the clearing of a set lock_flag in processor *B* by executing a PREFETCH_M or PREFETCH_MEN in *B*'s locked ranges of physical addresses.

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. See Section 5.5 for more information.

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 an 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_C, STQ_U, WH64x) 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 WH64x instruction on processor *A* to a region within the lock range of processor *B*, where the execution of the WH64x changes the contents of memory, causes the lock_flag on processor *B* to be cleared. If the WH64x 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:

STx_C Ra.mx,disp.ab(Rb.ab) !Memory format

Operation:

```
va ← {Rbv + SEXT(disp)}

CASE
  big_endian_data: va' ← va XOR 0002            ! STQ_C
  big_endian_data: va' ← va XOR 1002            ! STL_C
  little_endian_data: va' ← va                    ! STL_C
ENDCASE

IF lock_flag EQ 1 THEN
  (va')<31:0> ← Rav<31:0>                        ! STL_C
  (va')        ← Rav                              ! STQ_C
Ra ← lock_flag
lock_flag ← 0
```

Exceptions:

Access Violation
Fault on Write
Alignment
Translation Not Valid

Instruction mnemonics:

STL_C Store Longword from Register to Memory Conditional
STQ_C 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' is computed from va by inverting va<2> (bit 2 of the virtual address), but any memory management fault is reported for the original 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:

```
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:

STx Ra.rx,disp.ab(Rb.ab) !Memory format

Operation:

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

CASE

big_endian_data: va' $\leftarrow va \text{ XOR } 000_2$!STQ

big_endian_data: va' $\leftarrow va \text{ XOR } 100_2$!STL

big_endian_data: va' $\leftarrow va \text{ XOR } 110_2$!STW

big_endian_data: va' $\leftarrow va \text{ XOR } 111_2$!STB

little_endian_data: va' $\leftarrow va$

ENDCASE

(va') $\leftarrow Rav$!STQ

(va') $\langle 31:00 \rangle \leftarrow Rav\langle 31:0 \rangle$!STL

(va') $\langle 15:00 \rangle \leftarrow Rav\langle 15:0 \rangle$!STW

(va') $\langle 07:00 \rangle \leftarrow Rav\langle 07:0 \rangle$!STB

Exceptions:

Access Violation

Alignment

Fault on Write

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:

- The STB and STW instructions are supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 0. STB and STW are supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 0. Software emulation of STB and STW is significantly slower than hardware support.
- 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:

$$va \leftarrow \{ \{Rbv + \text{SEXT}(disp) \} \text{ AND NOT } 7 \}$$
$$(va)_{<63:0>} \leftarrow Rav_{<63:0>}$$

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:

- 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.
- For many implementations, the first-level (or only) I-cache is no bigger than a page (8 KB to 64 KB).
- 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

Mnemonic	Operation
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

Table 4–3: Control Instructions Summary (Continued)

Mnemonic	Operation
BR	Unconditional Branch
BSR	Branch to Subroutine
JMP	Jump
JSR	Jump to Subroutine
RET	Return from Subroutine
JSR_COROUTINE	Jump to Subroutine Return

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.

4.3.2 Unconditional Branch

Format:

BxR Ra.wq,disp.al !Branch format

Operation:

{update PC}
Ra ← PC
PC ← PC + {4*SEXT(disp)}

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:

```
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:

mnemonic Ra.wq,(Rb.ab),hint !Memory format

Operation:

{update PC}
va ← Rbv AND {NOT 3}
Ra ← PC
PC ← va

Exceptions:

None

Instruction mnemonics:

JMP	Jump
JSR	Jump to Subroutine
RET	Return from Subroutine
JSR_COROUTINE	Jump to Subroutine Return

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

disp<15:14>	Meaning	Predicted Target<15:0>	Prediction Stack Action
00	JMP	PC + {4*disp<13:0>}	–
01	JSR	PC + {4*disp<13:0>}	Push PC
10	RET	Prediction stack	Pop
11	JSR_COROUTINE	Prediction stack	Pop, push PC

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 Section A.2.3 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 Compaq software or Reserved to Compaq, as follows:

Encoding	Meaning
0000 ₁₆	Indicates non-procedure return
0001 ₁₆	Indicates procedure return
All other encodings are reserved to Compaq.	

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

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

There is no integer divide instruction. Division by a constant can be done by using UMULH; division by a variable can be done by using a subroutine. See Section A.4.2.

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:

$R_c \leftarrow \text{SEXT}((R_{av} + R_{bv})_{<31:0>})$

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 $R_{av}_{<31:0>} + R_{bv}_{<31:0>}$.

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:

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

Operation:

$Rc \leftarrow R_{av} + R_{bv}$

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:

SxADDQ Ra.rq,Rb.rq,Rc.wq !Operate format

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

Operation:

```
CASE
  S4ADDQ: Rc ← LEFT_SHIFT(Rav, 2) + Rbv
  S8ADDQ: Rc ← LEFT_SHIFT(Rav, 3) + Rbv
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

S4ADDQ	Scaled Add Quadword by 4
S8ADDQ	Scaled Add Quadword by 8

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
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:

CMPU _{xx}	Ra.rq,Rb.rq,Rc.wq	!Operate format
CMPU _{xx}	Ra.rq,#b.ib,Rc.wq	!Operate format

Operation:

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

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:

CTLZ Rb.rq,Rc.wq ! Operate format

Operation:

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

Exceptions:

None

Instruction mnemonics:

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.

Implementation Notes:

- The CTLZ instruction is supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 2. CTLZ is supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 2. Software emulation of CTLZ is significantly slower than hardware support.

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:

CTPOP Count Population

Qualifiers:

None

Description:

The number of ones in Rb is written to Rc. Ra must be R31.

Implementation Notes:

- The CTPOP instruction is supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 2. CTPOP is supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 2. Software emulation of CTPOP is significantly slower than hardware support.

4.4.9 Count Trailing Zero

Format:

CTTZ Rb.rq,Rc.wq ! Operate format

Operation:

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

Exceptions:

None

Instruction mnemonics:

CTTZ 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.

Implementation Notes:

- The CTTZ instruction is supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 2. CTTZ is supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 2. Software emulation of CTTZ is significantly slower than hardware support.

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:

$$Rc \leftarrow \text{SEXT} ((Rav * Rbv) < 31:0 >)$$

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 $Rav < 31:0 > * Rbv < 31:0 >$. On overflow, the proper sign extension of the least significant 32 bits of the true result is written to the destination register.

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

4.4.11 Quadword Multiply

Format:

MULQ Ra.rq,Rb.rq,Rc.wq !Operate format

MULQ Ra.Rq,#b.ib,Rc.wq !Operate format

Operation:

$Rc \leftarrow R_{av} * R_{bv}$

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:

$Rc \leftarrow \{Rav * U Rbv\}_{<127:64>}$

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: (result of UMULH) – Ra<63>*Rb – Rb<63>*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:

$$Rc \leftarrow \text{SEXT} ((Rav - Rbv) < 31:0 >)$$

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:

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

Operation:

```
CASE
  S4SUBL: Rc ← SEXT ( ((LEFT_SHIFT(Rav,2)) - Rbv)<31:0> )
  S8SUBL: Rc ← SEXT ( ((LEFT_SHIFT(Rav,3)) - Rbv)<31:0> )
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

S4SUBL	Scaled Subtract Longword by 4
S8SUBL	Scaled Subtract Longword 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 S4SUBL) or 8 (for 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:

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

Operation:

$Rc \leftarrow Rav - Rbv$

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), a borrow will occur.

4.4.16 Scaled Quadword Subtract

Format:

SxSUBQ Ra.rq,Rb.rq,Rc.wq !Operate format

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

Operation:

```
CASE
  S4SUBQ: Rc ← LEFT_SHIFT(Rav, 2) - Rbv
  S8SUBQ: Rc ← LEFT_SHIFT(Rav, 3) - Rbv
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

Mnemonic	Operation
AND	Logical Product
BIC	Logical Product with Complement
BIS	Logical Sum (OR)
EQV	Logical Equivalence (XORNOT)
ORNOT	Logical Sum with Complement
XOR	Logical Difference
CMOV _{xx}	Conditional Move Integer
SLL	Shift Left Logical
SRA	Shift Right Arithmetic
SRL	Shift Right Logical

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 a 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:

$Rc \leftarrow Rav \text{ AND } Rbv$!AND
$Rc \leftarrow Rav \text{ OR } Rbv$!BIS
$Rc \leftarrow Rav \text{ XOR } Rbv$!XOR
$Rc \leftarrow Rav \text{ AND } \{\text{NOT } Rbv\}$!BIC
$Rc \leftarrow Rav \text{ OR } \{\text{NOT } Rbv\}$!ORNOT
$Rc \leftarrow Rav \text{ XOR } \{\text{NOT } Rbv\}$!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:

CMOV _{xx}	Ra.rq,Rb.rq,Rc.wq	!Operate format
CMOV _{xx}	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:

$Rc \leftarrow \text{ARITH_RIGHT_SHIFT}(Rav, Rbv<5:0>)$

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

Alpha implementations that support the BWX extension provide the following instructions for loading, sign-extending, and storing bytes and words between a register and memory:

Instruction	Meaning	Described in Section
LDBU/LDWU	Load byte/word unaligned	4.2.2
SEXTB/SEXTW	Sign-extend byte/word	4.6.5
STB/STW	Store byte/word	4.2.6

The AMASK and IMPLVER instructions report whether a particular Alpha implementation supports the BWX extension. AMASK and IMPLVER are described in Sections 4.11.1 and 4.11.6, respectively, and in Appendix D.

LDBU and STB are the recommended way to perform byte load and store operations on Alpha implementations that support them; use them rather than the extract, insert, and mask byte instructions described in this section. In particular, the implementation examples in this section that illustrate byte operations are not appropriate for Alpha implementations that support the BWX extension – instead use the recommendations in Appendix A.

In addition to LDBU and STB, Alpha provides the instructions in Table 4–7 for operating on byte operands within registers.

Table 4–7: Byte-Within-Register Manipulation Instructions Summary

Mnemonic	Operation
CMPBGE	Compare Byte
EXTBL	Extract Byte Low
EXTWL	Extract Word Low
EXTLL	Extract Longword Low
EXTQL	Extract Quadword Low
EXTWH	Extract Word High
EXTLH	Extract Longword High
EXTQH	Extract Quadword High
INSBL	Insert Byte Low
INSWL	Insert Word Low
INSL	Insert Longword Low
INSQL	Insert Quadword Low
INSWH	Insert Word High
INSLH	Insert Longword High

Table 4–7: Byte-Within-Register Manipulation Instructions Summary

Mnemonic	Operation
INSQH	Insert Quadword High
MSKBL	Mask Byte Low
MSKWL	Mask Word Low
MSKLL	Mask Longword Low
MSKQL	Mask Quadword Low
MSKWH	Mask Word High
MSKLH	Mask Longword High
MSKQH	Mask Quadword High
SEXTB	Sign Extend Byte
SEXTW	Sign Extend Word
ZAP	Zero Bytes
ZAPNOT	Zero Bytes Not

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/equal/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
CMPBGE R31, R3, R6         ; Test for zeros in string1
XOR    R3, R4, R5          ; Test for all equal bytes
BNE    R6, DONE            ; Exit if a zero found
BEQ    R5, LOOP            ; Loop if all equal
DONE:  CMPBGE R31, R5, R5   ;
```

...

; At this point, R5 can be used to determine the first not-equal
; byte position (if any), and R6 can be used to determine the
; position of the terminating zero in string1 (if any).

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:

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

Operation:

```
CASE
  big_endian_data: Rbv' ← Rbv XOR 1112
  little_endian_data: Rbv' ← Rbv
ENDCASE

CASE
  EXTBL: byte_mask ← 0000 00012
  EXTWL: byte_mask ← 0000 00112
  EXTLL: byte_mask ← 0000 11112
  EXTQL: byte_mask ← 1111 11112
  EXTH:  byte_mask ← 0000 00012
  EXTHL: byte_mask ← 0000 00112
  EXTHLH: byte_mask ← 0000 11112
  EXTHLH: byte_mask ← 1111 11112
ENDCASE

CASE
  EXTxL:
    byte_loc ← Rbv'<2:0>*8
    temp ← RIGHT_SHIFT(Rav, byte_loc<5:0>)
    Rc ← BYTE_ZAP(temp, NOT(byte_mask) )
  EXTxH:
    byte_loc ← 64 - Rbv'<2:0>*8
    temp ← LEFT_SHIFT(Rav, byte_loc<5:0>)
    Rc ← BYTE_ZAP(temp, NOT(byte_mask) )
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

EXTBL	Extract Byte Low
EXTWL	Extract Word Low
EXTLL	Extract Longword Low
EXTQL	Extract Quadword Low
EXTH	Extract Word High
EXTHL	Extract Longword High
EXTHLH	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 D000
OR     R2, R1, R1           ; R1 = 0000 DCBA
ADDL   R31, R1, R1          ; R1 = ssss DCBA
```

For software that is not designed to use the BWX extension, 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 extension, 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+1+1(R11)       ; R3<2:0> = 5+1+1 = 7
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 extension, 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 extension, 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 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:

```
LDL    R1, 8(R3)           ; R1 = ssss BAxx
                               ; Faults if R3 is not longword aligned
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:

```
LDL    R1, 8(R3)           ; R1 = ssss BAxx
                               ; Faults if R3 is not longword aligned
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:

```
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:

```
LDQ_U  R1, X(R11)         ; Ignores va<2:0>, R1 = xxxxxxABC
LDQ_U  R2, X+7(R11)       ; Ignores va<2:0>, R2 = DEFHyyy
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.

4.6.3 Byte Insert

Format:

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

Operation:

```
CASE
  big_endian_data: Rbv' ← Rbv XOR 1112
  little_endian_data: Rbv' ← Rbv
ENDCASE

CASE
  INSBL: byte_mask ← 0000 0000 0000 00012
  INSWL: byte_mask ← 0000 0000 0000 00112
  INSLx: byte_mask ← 0000 0000 0000 11112
  INSQx: 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:

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

Operation:

```
CASE
  big_endian_data: Rbv'← Rbv XOR 1112
  little_endian_data: Rbv'← Rbv
ENDCASE

CASE
  MSKBL: byte_mask ← 0000 0000 0000 00012
  MSKWx: byte_mask ← 0000 0000 0000 00112
  MSKLx: byte_mask ← 0000 0000 0000 11112
  MSKQx: byte_mask ← 0000 0000 1111 11112
ENDCASE
byte_mask ← LEFT_SHIFT(byte_mask, Rbv'<2:0>)

CASE
  MSKxL:
    Rc ← BYTE_ZAP(Rav, byte_mask<7:0>)
  MSKxH:
    Rc ← BYTE_ZAP(Rav, byte_mask<15:8>)
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

MSKBL	Mask Byte Low
MSKWL	Mask Word Low
MSKLL	Mask Longword Low
MSKQL	Mask Quadword Low
MSKWH	Mask Word High
MSKLH	Mask Longword High
MSKQH	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 $(ea \bmod 8) = 5$, 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
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 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)          ;
```

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.

Implementation Note:

The SEXTB and SEXTW instructions are supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 0. SEXTB and SEXTW are supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 0. Software emulation of SEXTB and SEXTW is significantly slower than hardware support.

4.6.6 Zero Bytes

Format:

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

Operation:

```
CASE
  ZAP:
    Rc ← BYTE_ZAP(Rav, Rbv<7:0>)

  ZAPNOT:
    Rc ← BYTE_ZAP(Rav, NOT Rbv<7:0>)
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

ZAP	Zero Bytes
ZAPNOT	Zero Bytes Not

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 (*not* including loads or stores) that yield an F_floating or G_floating zero result must materialize a true zero.

4.7.1 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.

Single-precision operations applied to canonical single-precision values give single-precision results. Floating-point operations applied to non-canonical single-precision operands give UNPREDICTABLE results.

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>. Floating-point operations applied to longword integer operations, where the operand register contains a non-zero value in bits <28:0>, give UNPREDICTABLE results.

4.7.2 Subsets and 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 (DNOD, DZED, INED, INV, OVFD, and UNFD) and Denormal Control flags (DNZ and UNDZ) in the FPCR:

- If a Trap Disable flag is not implemented, then the corresponding trap occurs as usual.
- If DNZ is not implemented, then any IEEE operation with a denormal input must take an Invalid Operation Trap.
- If UNDZ is not implemented, then any IEEE operation that includes a /S qualifier that underflows must take an Underflow Trap.
- If DZED is implemented, then IEEE division of 0/0 must be treated as an invalid operation instead of a division by zero.

Any unimplemented bits in the FPCR are read as zero and ignored when set.

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 infinities, 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 finite representable number A , $A + 1$ LSB is the next larger representative number, and $A + \frac{1}{2}$ LSB is exactly halfway between A and the next larger representable number. For a positive representable number A whose fraction field is not all zeros, $A - 1$ LSB is the next smaller representable number, and $A - \frac{1}{2}$ LSB is exactly halfway between A and the next smaller 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 an 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:

Sign	Exponent	Fraction	VAX Meaning	VAX Finite	IEEE Meaning	IEEE Finite
x	All-1's	Non-zero	Finite	Yes	+/-NaN	No
x	All-1's	0	Finite	Yes	+/-Infinity	No
0	0	Non-zero	Dirty zero	No	+Denormal	No
1	0	Non-zero	Resv. operand	No	-Denormal	No
0	0	0	True zero	Yes	+0	Yes
1	0	0	Resv. operand	No	-0	Yes
x	Other	x	Finite	Yes	Finite	Yes

The values of MIN and MAX for each of the five floating-point data formats are:

Data Format	MIN	MAX
F_floating	$2^{*-127} * 0.5$ (0.293873588e-38)	$2^{*127} * (1.0 - 2^{*-24})$ (1.7014117e38)
G_floating	$2^{*-1023} * 0.5$ (0.5562684646268004e-308)	$2^{*1023} * (1.0 - 2^{*-53})$ (0.89884656743115785407e308)
S_floating	$2^{*-126} * 1.0$ (1.17549435e-38)	$2^{*127} * (2.0 - 2^{*-23})$ (3.40282347e38)
T_floating	$2^{*-1022} * 1.0$ (2.2250738585072013e-308)	$2^{*1023} * (2.0 - 2^{*-52})$ (1.7976931348623158e308)
X_floating	$2^{*-16382} * 1.0$ (See below [†])	$2^{*16383} * (2.0 - 2^{*-112})$ (See below [‡])

[†] (1.18973149535723176508575932662800702e4932)

[‡] (3.36210314311209350626267781732175260e-4932)

4.7.5 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 $\geq \text{MAX} + 1/2 \text{ LSB}$ in magnitude to an overflow; maps non-zero true results $< \text{MIN} - 1/4 \text{ LSB}$ 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 $\geq \text{MAX} + 1 \text{ LSB}$ in magnitude to an overflow; maps non-zero true results $< \text{MIN}$ 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 instruction. 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 4.7.8). 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 non-zero 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 positive true results $> \text{MAX}$ to an overflow; maps negative true results $< -\text{MAX} - 1 \text{ LSB}$ to an overflow; maps 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 positive true results $> \text{MAX} + 1 \text{ LSB}$ to an overflow; maps negative true results $< -\text{MAX}$ 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 4.7.8.

The following tables summarize the floating-point rounding modes:

VAX Rounding Mode	Instruction Notation
Normal rounding	(No qualifier)
Chopped	/C

IEEE Rounding Mode	Instruction Notation
Normal rounding	(No qualifier)
Dynamic rounding	/D
Plus infinity	/D and ensure that FPCR<DYN> = '11'
Minus infinity	/M
Chopped	/C

4.7.6 Computational Models

The Alpha architecture provides a choice of floating-point computational models.

There are two computational models available on systems that implement the VAX floating-point subset:

- VAX-format arithmetic with precise exceptions
- High-performance VAX-format arithmetic

There are three computational models available on systems that implement the IEEE floating-point subset:

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

4.7.6.1 VAX-Format Arithmetic with Precise Exceptions

This model provides floating-point arithmetic that is fully compatible with the floating-point arithmetic provided by the VAX architecture. It provides support for VAX non-finites and gives precise exceptions.

This model is implemented by using VAX floating-point instructions with the /S, /SU, and /SV trap qualifiers. Each instruction can determine whether it also takes an exception on underflow or integer overflow. The performance of this model depends on how often computations involve non-finite operands. Performance also depends on how an Alpha system chooses to trade off implementation complexity between hardware and operating system completion handlers (see Section 4.7.7.3).

4.7.6.2 High-Performance VAX-Format Arithmetic

This model provides arithmetic operations on VAX finite numbers. An imprecise arithmetic trap is generated by any operation that involves non-finite numbers, floating overflow, and divide-by-zero exceptions.

This model is implemented by using VAX floating-point instructions with a trap qualifier other than /S, /SU, or /SV. Each instruction can determine whether it also traps on underflow or integer overflow. This model does not require the overhead of an operating system completion handler and can be the faster of the two VAX models.

4.7.6.3 IEEE-Compliant Arithmetic

This model provides floating-point arithmetic that fully complies with the IEEE Standard for Binary Floating-Point Arithmetic. It provides all of the exception status flags that are in the standard. It provides a default where all traps and faults are disabled and where IEEE non-finite values are used in lieu of exceptions.

Alpha operating systems provide additional mechanisms that allow the user to specify dynamically which exception conditions should trap and which should proceed without trapping. The operating systems also include mechanisms that allow alternative handling of denormal values. See Appendix B and the appropriate operating system documentation for a description of these mechanisms.

This model is implemented by using IEEE floating-point instructions with the /SUI or /SVI trap qualifiers. The performance of this model depends on how often computations involve inexact results and non-finite operands and results. Performance also depends on how the Alpha system chooses to trade off implementation complexity between hardware and operating system completion handlers (see Section 4.7.7.3). This model provides acceptable performance on Alpha systems that implement the inexact disable (INED) bit in the FPCR. Performance may be slow if the INED bit is not implemented.

4.7.6.4 IEEE-Compliant Arithmetic Without Inexact Exception

This model is similar to the model in Section 4.7.6.3, except this model does not signal inexact results either by the inexact status flag or by trapping. Combining routines that are compiled with this model and routines that are compiled with the model in Section 4.7.6.3 can give an application better control over testing when an inexact operation will affect computational accuracy.

This model is implemented by using IEEE floating-point instructions with the /SU or /SV trap qualifiers. The performance of this model depends on how often computations involve non-finite operands and results. Performance also depends on how an Alpha system chooses to trade off implementation complexity between hardware and operating system completion handlers (see Section 4.7.7.3).

4.7.6.5 High-Performance IEEE-Format Arithmetic

This model provides arithmetic operations on IEEE finite numbers and notifies applications of all exceptional floating-point operations. An imprecise arithmetic trap is generated by any operation that involves non-finite numbers, floating overflow, divide-by-zero, and invalid operations. Underflow results are set to zero. Conversion to integer results that overflow are set to the low-order bits of the integer value.

This model is implemented by using IEEE floating-point instructions with a trap qualifier other than /SU, /SV, /SUI, or /SVI. Each instruction can determine whether it also traps on underflow or integer overflow. This model does not require the overhead of an operating system completion handler and can be the fastest of the three IEEE models.

4.7.7 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
- Inexact result
- Integer overflow (conversion to integer only)

4.7.7.1 VAX Trapping Modes

This section describes the characteristics of the four VAX trapping modes, which are summarized in Table 4–8.

When no trap mode is specified (the default):

- Arithmetic is performed on VAX finite numbers.
- Operations give imprecise traps whenever the following occur:
 - an operand is a non-finite number
 - a floating overflow
 - a divide-by-zero
- Traps are imprecise and it is not always possible to determine which instruction triggered a trap or the operands of that instruction.
- An underflow produces a zero result without trapping.
- A conversion to integer that overflows uses the low-order bits of the integer as the result without trapping.
- The result of any operation that traps is UNPREDICTABLE.

When /U or /V mode is specified:

- Arithmetic is performed on VAX finite numbers.
- Operations give imprecise traps whenever the following occur:
 - an operand is a non-finite number
 - an underflow
 - an integer overflow
 - a floating overflow
 - a divide-by-zero
- Traps are imprecise and it is not always possible to determine which instruction triggered a trap or the operands of that instruction.
- An underflow trap produces a zero result.
- A conversion to integer trapping with an integer overflow produces the low-order bits of the integer value.
- The result of any other operation that traps is UNPREDICTABLE.

When /S mode is specified:

- Arithmetic is performed on all VAX values, both finite and non-finite.
- A VAX dirty zero is treated as zero.

- Exceptions are signaled for:
 - a VAX reserved operand, which generates an invalid operation exception
 - a floating overflow
 - a divide-by-zero
- Exceptions are precise and an application can locate the instruction that caused the exception, along with its operand values. See Section 4.7.7.3.
- An operation that underflows produces a zero result without taking an exception.
- A conversion to integer that overflows uses the low-order bits of the integer as the result, without taking an exception.
- When an operation takes an exception, the result of the operation is UNPREDICTABLE.

When /SU or /SV mode is specified:

- Arithmetic is performed on all VAX values, both finite and non-finite.
- A VAX dirty zero is treated as zero.
- Exceptions are signaled for:
 - a VAX reserved operand, which generates an invalid operation exception
 - an underflow
 - an integer overflow
 - a floating overflow
 - a divide-by-zero
- Exceptions are precise and an application can locate the instruction that caused the exception, along with its operand values. See Section 4.7.7.3.
- An underflow exception produces a zero.
- A conversion to integer exception with integer overflow produces the low-order bits of the integer value.
- The result of any other operation that takes an exception is UNPREDICTABLE.

A summary of the VAX trapping modes, instruction notation, and their meaning follows in Table 4–8:

Table 4–8: VAX Trapping Modes Summary

Trap Mode	Notation	Meaning
Underflow disabled	No qualifier	Imprecise
	/S	Precise exception completion
Underflow enabled	/U	Imprecise
	/SU	Precise exception completion
Integer overflow disabled	No qualifier	Imprecise
	/S	Precise exception completion
Integer overflow enabled	/V	Imprecise
	/SV	Precise exception completion

4.7.7.2 IEEE Trapping Modes

This section describes the characteristics of the four IEEE trapping modes, which are summarized in Table 4–9.

When no trap mode is specified (the default):

- Arithmetic is performed on IEEE finite numbers.
- Operations give imprecise traps whenever the following occur:
 - an operand is a non-finite number
 - a floating overflow
 - a divide-by-zero
 - an invalid operation
- Traps are imprecise, and it is not always possible to determine which instruction triggered a trap or the operands of that instruction.
- An underflow produces a zero result without trapping.
- A conversion to integer that overflows uses the low-order bits of the integer as the result without trapping.
- When an operation traps, the result of the operation is UNPREDICTABLE.

When /U or /V mode is specified :

- Arithmetic is performed on IEEE finite numbers.
- Operations give imprecise traps whenever the following occur:
 - an operand is a non-finite number
 - an underflow
 - an integer overflow
 - a floating overflow
 - a divide-by-zero
 - an invalid operation
- Traps are imprecise, and it is not always possible to determine which instruction triggered a trap or the operands of that instruction.
- An underflow trap produces a zero.
- A conversion to integer trap with an integer overflow produces the low-order bits of the integer.
- The result of any other operation that traps is UNPREDICTABLE.

When /SU or /SV mode is specified:

- Arithmetic is performed on all IEEE values, both finite and non-finite.
- Alpha systems support all IEEE features except inexact exception (which requires /SUI or /SVI):
 - The IEEE standard specifies a default where exceptions do not fault or trap. In combination with the FPCR, this mode allows disabling exceptions and producing IEEE compliant nontrapping results. See Sections 4.7.7.10 and 4.7.7.11.
 - Each Alpha operating system provides a way to optionally signal IEEE floating-point exceptions. This mode enables the IEEE status flags that keep a record of each exception that is encountered. An Alpha operating system uses the IEEE floating-point control (FP_C) quadword, described in Appendix B, to maintain the IEEE status flags and to enable calls to IEEE user signal handlers.
- Exceptions signaled in this mode are precise and an application can locate the instruction that caused the exception, along with its operand values. See Section 4.7.7.3.

When /SUI or /SVI mode is specified:

- Arithmetic is performed on all IEEE values, both finite and non-finite.
- Inexact exceptions are supported, along with all the other IEEE features supported by the /SU or /SV mode.

A summary of the IEEE trapping modes, instruction notation, and their meaning follows in Table 4–9.

Table 4–9 Summary of IEEE Trapping Modes

Trap Mode	Notation	Meaning
Underflow disabled and inexact disabled	No qualifier	Imprecise
Underflow enabled and inexact disabled	/U	Imprecise
	/SU	Precise exception completion
Underflow enabled and inexact enabled	/SUI	Precise exception completion
Integer overflow disabled and inexact disabled	No qualifier	Imprecise
Integer overflow enabled and inexact disabled	/V	Imprecise
	/SV	Precise exception completion
Integer overflow enabled and inexact enabled	/SVI	Precise exception completion

4.7.7.3 Arithmetic Trap Completion

Because floating-point instructions may be pipelined, the trap PC can be an arbitrary number of instructions past the one triggering the trap. Those instructions that are executed after the trigger instruction of an arithmetic trap are collectively referred to as the *trap shadow* of the trigger instruction.

Marking floating-point instructions for exception completion with any valid qualifier combination that includes the /S qualifier enables the completion of the triggering instruction. For any instruction so marked, the output register for the triggering instruction cannot also be one of the input registers, so that an input register cannot be overwritten and the input value is available after a trap occurs.

See Section B.2 for more information.

The AMASK instruction reports how the arithmetic trap should be completed:

- If AMASK does not clear feature mask bit 9, floating-point traps are imprecise. Exception completion requires that generated code must obey the trap shadow rules in Section 4.7.7.3.2, with a trap shadow length as described in Section 4.7.7.3.2.

- If AMASK clears feature mask bit 9, the hardware implements precise floating-point traps. If the instruction has any valid qualifier combination that includes /S, the trap PC points to the instruction that immediately follows the instruction that triggered the trap. The trap shadow contains zero instructions; exception completion does not require that the generated code follow the conditions in Section • and the length rules in Section 4.7.7.3.2.

4.7.7.3.1 Trap Shadow Rules

For an operating system (OS) completion handler to complete non-finite operands and exceptions, the following conditions must hold.

Conditions 1 and 2, below, allow an OS completion handler to locate the trigger instruction by doing a linear scan backwards from the trap PC while comparing destination registers in the trap shadow with the registers that are specified in the register write mask parameter to the arithmetic trap.

Condition 3 allows an OS completion handler to emulate the trigger instruction with its original input operand values.

Condition 4 allows the handler to re-execute instructions in the trap shadow with their original operand values.

Condition 5 prevents any unusual side effects that would cause problems on repeated execution of the instructions in the trap shadow.

Conditions:

1. The destination register of the trigger instruction may not be used as the destination register of any instruction in the trap shadow.
2. The trap shadow may not include any branch or jump instructions.
3. An instruction in the trap shadow may not modify an input to the trigger instruction.
4. The value in a register or memory location that is used as input to some instruction in the trap shadow may not be modified by a subsequent instruction in the trap shadow unless that value is produced by an earlier instruction in the trap shadow.
5. The trap shadow may not contain any instructions with side effects that interact with earlier instructions in the trap shadow or with other parts of the system. Examples of operations with prohibited side effects are:
 - Modifications of the stack pointer or frame pointer that can change the accessibility of stack variables and the exception context that is used by earlier instructions in the trap shadow.
 - Modifications of volatile values and access to I/O device registers.
 - If order of exception reporting is important, taking an arithmetic trap by an integer instruction or by a floating-point instruction that does not include a /S qualifier, either of which can report exceptions out of order.

An instruction may be in the trap shadows of multiple instructions that include a /S qualifier. That instruction must obey all conditions for all those trap shadows. For example, the destination register of an instruction in multiple trap shadows must be different than the destination registers of each possible trigger instruction.

4.7.7.3.2 Trap Shadow Length Rules

The trap shadow length rules in Table 4–11 apply only to those floating-point instructions with any valid qualifier combination that includes a /S trap qualifier. Further, the instruction to which the trap shadow extends is not part of the trap shadow and that instruction is not executed prior to the arithmetic trap that is signaled by the trigger instruction.

Implementation notes:

- On Alpha implementations for which the IMPLVER instruction returns the value 0, the trap shadow of an instruction may extend after the result is consumed by a floating-point STx instruction. On all other implementations, the trap shadow ends when a result is consumed.
- Because Alpha implementations need not execute instructions that have R31 or F31 as the destination operand, instructions with such a destination should not be thought to end a trap shadow.

Table 4–10 Trap Shadow Length Rules

Floating-Point Instruction Group	Trap Shadow Extends Until Any of the Following Occurs:
Floating-point operate (except DIVx and SQRTx)	<ul style="list-style-type: none"> • Encountering a CALL_PAL, EXCB, or TRAPB instruction. • The result is consumed by any instruction except floating-point STx. • The fourth instruction[†] after the result is consumed by a floating-point STx instruction. Or, following the floating-point STx of the result, the result of a LDx that loads the stored value is consumed by any instruction. • The result of a subsequent floating-point operate instruction is consumed by any instruction except floating-point STx. • The second instruction[†] after the result of a subsequent floating-point operate instruction is consumed by a floating-point STx instruction. • The result of a subsequent floating-point DIVx or SQRTx instruction is consumed by any instruction.
Floating-point DIVx	<ul style="list-style-type: none"> • Encountering a CALL_PAL, EXCB, or TRAPB instruction. • The result is consumed by any instruction except floating-point STx. • The fourth instruction[†] after the result is consumed by a floating-point STx instruction. Or, following the floating-point STx of the result, the result of a LDx that loads the stored value is consumed by any instruction. • The result of a subsequent floating-point DIVx is consumed by any instruction.
Floating-point SQRTx	<ul style="list-style-type: none"> • Encountering a CALL_PAL, EXCB, or TRAPB instruction. • The result is consumed by any instruction. • The result of a subsequent SQRTx instruction is consumed by any instruction.

[†] The length of four instructions is a conservative estimate of how far the trap shadow may extend past a consuming floating-point STx instruction. The length of two instructions is a conservative estimate of how far the trap shadow may extend after a subsequent floating-point operate instruction is consumed by a floating-point STx instruction. Compilers can make a more precise estimate by consulting the Hardware Reference Manual for a particular processor at ftp.compaq.com/pub/products/alphaCPU-docs.

4.7.7.4 Invalid Operation (INV) Arithmetic Trap

An invalid operation arithmetic trap is signaled if an operand is a non-finite number or if an operand is invalid for the operation to be performed. (Note that CMPTxy does not trap on plus or minus infinity.) Invalid operations are:

- Any operation on a signaling NaN.
- Addition of unlike-signed infinities or subtraction of like-signed infinities, such as (+infinity + -infinity) or (+infinity - +infinity).
- Multiplication of 0*infinity.
- IEEE division of 0/0 or infinity/infinity.
- Conversion of an infinity or NaN to an integer.
- CMPTLE or CMPTLT when either operand is a NaN.
- SQRTx of a negative non-zero number.

The instruction cannot disable the trap and, if the trap occurs, an UNPREDICTABLE value is stored in the result register. However, under some conditions, the FPCR can dynamically disable the trap, as described in Section 4.7.7.10, producing a correct IEEE result, as described in Section 4.7.10.

IEEE-compliant system software must also supply an invalid operation indication to the user for x REM 0 and for conversions to integer that take an integer overflow trap.

If an implementation does not support the DZED (division by zero disable) bit, it may respond to the IEEE 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.

An implementation may choose not to take an INV trap for a valid IEEE operation that involves denormal operands if:

- The instruction is modified by any valid qualifier combination that includes the /S (exception completion) qualifier.
- The implementation supports the DNZ (denormal operands to zero) bit and DNZ is set.
- The instruction produces the result and exceptions required by Section 4.7.10, as modified by the DNZ bit described in Section 4.7.7.11.

An implementation may choose not to take an INV trap for a valid IEEE operation that involves denormal operands, and direct hardware implementation of denormal arithmetic is permitted if:

- The instruction is modified by any valid qualifier combination that includes the /S (exception completion) qualifier.
- The implementation supports both the DNOD (denormal operand exception disable) bit and the DNZ (denormal operands to zero) bit and DNOD is set while DNZ is clear.
- The instruction produces the result and exceptions required by Section 4.7.10, possibly modified by the UDNZ bit described in Section 4.7.7.11.

Regardless of the setting of the INVD (invalid operation disable) bit, the implementation may choose not to trap on valid operations that involve quiet NaNs and infinities as operands for IEEE instructions that are modified by any valid qualifier combination that includes the /S (exception completion) qualifier.

4.7.7.5 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 and, if the trap occurs, an UNPREDICTABLE value is stored in the result register. However, under some conditions, the FPCR can dynamically disable the trap, as described in Section 4.7.7.10, producing a correct IEEE result, as described in Section 4.7.10.

If an implementation does not support the DZED (division by zero disable) bit, it may respond to the IEEE 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.

4.7.7.6 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 and, if the trap occurs, an UNPREDICTABLE value is stored in the result register. However, under some conditions, the FPCR can dynamically disable the trap, as described in Section 4.7.7.10, producing a correct IEEE result, as described in Section 4.7.10.

4.7.7.7 Underflow (UNF) Arithmetic Trap

Section 4.7.5 defines conditions under which an underflow occurs.

Note:

The Alpha hardware definition underflow differs from the IEEE definition in that the Alpha definition does not depend on whether the result is inexact. Alpha provides IEEE compliant underflow handling by means of a software completion handler, which is described in Appendix B.

If an underflow trap occurs, a true zero (64 bits of zero) is always stored in the result register. In the case of an IEEE operation that takes an underflow arithmetic trap, a true zero is stored even if the result after rounding 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. However, under some conditions, the FPCR can dynamically disable the trap, as described in Section 4.7.7.10, producing the result described in Section 4.7.10, as modified by the UNDZ bit described in Section 4.7.7.11.

4.7.7.8 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. However, under some conditions, the FPCR can dynamically disable the trap; see Section 4.7.7.10 for information.

4.7.7.9 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.

4.7.7.10 IEEE Floating-Point Trap Disable Bits

In the case of IEEE exception completion modes, any of the traps described in Sections through 4.7.7.9 may be disabled by setting the appropriate trap disable bit in the FPCR. The trap disable bits only affect the IEEE trap modes when the instruction is modified by any valid qualifier combination that includes the /S (exception completion) qualifier. The trap disable bits (DNOD, DZED, INED, INV D, OVFD, and UNFD) 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 denormal control bits. If the implementation is unable to calculate the required result, it ignores the trap disable bit and signals a trap as usual.

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

4.7.7.11 IEEE Denormal Control Bits

In the case of IEEE exception completion modes, the handling of denormal operands and results is controlled by the DNZ and UN DZ bits in the FPCR. These denormal control bits only affect denormal handling by IEEE instructions that are modified by any valid qualifier combination that includes the /S (exception completion) qualifier.

The denormal control bits apply only to the IEEE operate instructions – ADD, SUB, MUL, DIV, SQRT, CMPxx, and CVT with floating-point source operand.

If both the UNFD (underflow disable) bit and the UN DZ (underflow to zero) bit are set in the FPCR, the implementation sets the result of an underflow operation to a true zero result. The zeroing of a denormal result by UN DZ must also be treated as an inexact result.

If the DNZ (denormal operands to zero) bit is set in the FPCR, the implementation treats each denormal operand as if it were a signed zero value. The source operands in the register are not changed. If DNZ is set, IEEE operations with any valid qualifier combination that includes a /S qualifier signal arithmetic traps as if any denormal operand were zero; that is, with DNZ set:

- An IEEE operation with a denormal operand never generates an overflow, underflow, or inexact result arithmetic trap.

- Dividing by a denormal operand is a division by zero or invalid operation as appropriate.
- Multiplying a denormal by infinity is an invalid operation.
- A SQRT of a negative denormal produces a -0 instead of an invalid operation.
- A denormal operand, treated as zero, does not take the denormal operand exception trap controlled by the DNOD bit in the FPCR.

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

4.7.8 Floating-Point Control Register (FPCR)

When an IEEE floating-point operate instruction specifies dynamic mode (*/D*) in its function field (function field bits $\langle 12:11 \rangle = 11$), 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 exception bits found in the exception summary quadword that is pushed for arithmetic traps. However, for each instruction, these exception 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 is still 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.

Section 4.7.2 allows certain of the FPCR bits to be subsetted.

The format of the FPCR is shown in Figure 4–1 and described in Table 4–11.

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

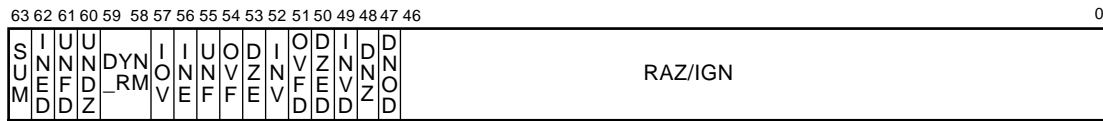


Table 4–11 Floating-Point Control Register (FPCR) Bit Descriptions

Bit	Description (Meaning When Set)										
63	Summary Bit (SUM). Records bitwise OR of FPCR exception bits. Equal to FPCR<57 56 55 54 53 52>. The summary bit is not directly modified by writes to bit 63 of the FPCR, but is indirectly modified by changes to FPCR bits 57–52.										
62	Inexact Disable (INED) [†] . Suppress INE trap and place correct IEEE nontrapping result in the destination register.										
61	Underflow Disable (UNFD) [†] . If the implementation is capable of producing the correct IEEE nontrapping underflow result, suppress the UNF trap and place the appropriate result value in the destination register. The correct result value is determined according to the value of the UNF bit.										
60	Underflow to Zero (UNDZ) [†] . Determines the result value in the destination register when an underflow trap is disabled. When set, the non-trapping underflow result value is a true zero (64 bits of zero); when clear, the non-trapping underflow result value is the non-trapping result (denorm, +0 or –0) specified in the IEEE standard.										
59–58	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:										
	<table border="1"> <thead> <tr> <th>DYN</th> <th>IEEE Rounding Mode Selected</th> </tr> </thead> <tbody> <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> </tbody> </table>	DYN	IEEE Rounding Mode Selected	00	Chopped rounding mode	01	Minus infinity	10	Normal rounding	11	Plus infinity
DYN	IEEE Rounding Mode Selected										
00	Chopped rounding mode										
01	Minus infinity										
10	Normal rounding										
11	Plus infinity										
57	Integer Overflow (IOV). A CVTGQ, CVTTQ, or CVTQL instruction overflowed the destination precision.										
56	Inexact Result (INE). A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.										
55	Underflow (UNF). A floating arithmetic or conversion operation underflowed the destination exponent.										
54	Overflow (OVF). A floating arithmetic or conversion operation overflowed the destination exponent.										
53	Division by Zero (DZE). An attempt was made to perform a floating divide operation with a divisor of zero.										
52	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.										

Table 4–11 Floating-Point Control Register (FPCR) Bit Descriptions (Continued)

Bit	Description (Meaning When Set)
51	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 results.
50	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 results.
49	Invalid Operation Disable (INVVD) [†] . Suppress INV trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping results.
48	Denormal Operands to Zero (DNZ) [†] . Treat all denormal operands as a signed zero value with the same sign as the denormal.
47	Denormal Operand Exception Disable (DNOD) [†] . Suppress INV trap for valid operations that involve denormal operand values and place the correct IEEE nontrapping result in the destination register if the implementation is capable of processing the denormal operand. If the result of the operation underflows, the correct result is determined according to the value of the UNDZ bit. If DNZ is set, DNOD has no effect because a denormal operand is treated as having a zero value instead of a denormal value.
46–0	Reserved. Read as Zero. Ignored when written.

[†] Bit only has meaning for IEEE instructions when any valid qualifier combination that includes exception completion (/S) is specified.

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 4.7.8.1.

FPCR and the instructions to access it are required for an implementation that supports floating-point (see Section 4.7.8). 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 operating system even if that system does not support floating-point.

4.7.8.1 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>.

4.7.8.2 Default Values of the FPCR

Processor initialization leaves the value of FPCR UNPREDICTABLE.

Software Note:

Compaq software should initialize FPCR<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.

4.7.8.3 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 qualifiers. 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. See Appendix B.

When such a scheme is being used, the trap disable bits and denormal control 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.

Compaq 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 exception 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.

4.7.9 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 4-12 and for VAX floating-point in Table 4-13. Function codes for the independent floating-point instructions, those with opcode 17_{16} , 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

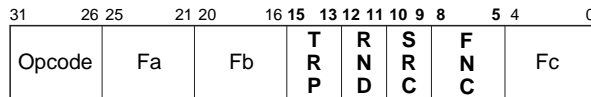


Table 4–12 IEEE Floating-Point Function Field Bit Summary

Bits	Field	Meaning [†]																																
15–13	TRP	Trapping modes: <table border="0"> <tr> <td style="text-align: right;">Contents</td> <td>Meaning for Opcodes 14₁₆ and 16₁₆</td> </tr> <tr> <td>000</td> <td>Imprecise (default)</td> </tr> <tr> <td>001</td> <td>Underflow enable (/U) — floating-point output</td> </tr> <tr> <td></td> <td>Integer overflow enable (/V) — integer output</td> </tr> <tr> <td>010</td> <td>UNPREDICTABLE for opcode 16₁₆ instructions</td> </tr> <tr> <td></td> <td>Reserved for opcode 14₁₆ instructions</td> </tr> <tr> <td>011</td> <td>UNPREDICTABLE for opcode 16₁₆ instructions</td> </tr> <tr> <td></td> <td>Reserved for opcode 14₁₆ instructions</td> </tr> <tr> <td>100</td> <td>UNPREDICTABLE for opcode 16₁₆ instructions</td> </tr> <tr> <td></td> <td>Reserved for opcode 14₁₆ instructions</td> </tr> <tr> <td>101</td> <td>/SU — floating-point output</td> </tr> <tr> <td></td> <td>/SV — integer output</td> </tr> <tr> <td>110</td> <td>UNPREDICTABLE for opcode 16₁₆ instructions</td> </tr> <tr> <td></td> <td>Reserved for opcode 14₁₆ instructions</td> </tr> <tr> <td>111</td> <td>/SUI — floating-point output</td> </tr> <tr> <td></td> <td>/SVI — integer output</td> </tr> </table>	Contents	Meaning for Opcodes 14₁₆ and 16₁₆	000	Imprecise (default)	001	Underflow enable (/U) — floating-point output		Integer overflow enable (/V) — integer output	010	UNPREDICTABLE for opcode 16 ₁₆ instructions		Reserved for opcode 14 ₁₆ instructions	011	UNPREDICTABLE for opcode 16 ₁₆ instructions		Reserved for opcode 14 ₁₆ instructions	100	UNPREDICTABLE for opcode 16 ₁₆ instructions		Reserved for opcode 14 ₁₆ instructions	101	/SU — floating-point output		/SV — integer output	110	UNPREDICTABLE for opcode 16 ₁₆ instructions		Reserved for opcode 14 ₁₆ instructions	111	/SUI — floating-point output		/SVI — integer output
Contents	Meaning for Opcodes 14₁₆ and 16₁₆																																	
000	Imprecise (default)																																	
001	Underflow enable (/U) — floating-point output																																	
	Integer overflow enable (/V) — integer output																																	
010	UNPREDICTABLE for opcode 16 ₁₆ instructions																																	
	Reserved for opcode 14 ₁₆ instructions																																	
011	UNPREDICTABLE for opcode 16 ₁₆ instructions																																	
	Reserved for opcode 14 ₁₆ instructions																																	
100	UNPREDICTABLE for opcode 16 ₁₆ instructions																																	
	Reserved for opcode 14 ₁₆ instructions																																	
101	/SU — floating-point output																																	
	/SV — integer output																																	
110	UNPREDICTABLE for opcode 16 ₁₆ instructions																																	
	Reserved for opcode 14 ₁₆ instructions																																	
111	/SUI — floating-point output																																	
	/SVI — integer output																																	

Table 4–12 IEEE Floating-Point Function Field Bit Summary (Continued)

Bits	Field	Meaning [†]																																																			
12–11	RND	Rounding modes:																																																			
		<table border="0"> <thead> <tr> <th>Contents</th> <th>Meaning for Opcodes 16₁₆ and 14₁₆</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Chopped (/C)</td> </tr> <tr> <td>01</td> <td>Minus infinity (/M)</td> </tr> <tr> <td>10</td> <td>Normal (default)</td> </tr> <tr> <td>11</td> <td>Dynamic (/D)</td> </tr> </tbody> </table>	Contents	Meaning for Opcodes 16 ₁₆ and 14 ₁₆	00	Chopped (/C)	01	Minus infinity (/M)	10	Normal (default)	11	Dynamic (/D)																																									
Contents	Meaning for Opcodes 16 ₁₆ and 14 ₁₆																																																				
00	Chopped (/C)																																																				
01	Minus infinity (/M)																																																				
10	Normal (default)																																																				
11	Dynamic (/D)																																																				
10–9	SRC	Source datatype:																																																			
		<table border="0"> <thead> <tr> <th>Contents</th> <th>Meaning for Opcode 16₁₆</th> <th>Meaning for Opcode 14₁₆</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>S_floating</td> <td>S_floating</td> </tr> <tr> <td>01</td> <td>Reserved</td> <td>Reserved</td> </tr> <tr> <td>10</td> <td>T_floating</td> <td>T_floating</td> </tr> <tr> <td>11</td> <td>Q_fixed</td> <td>Reserved</td> </tr> </tbody> </table>	Contents	Meaning for Opcode 16 ₁₆	Meaning for Opcode 14 ₁₆	00	S_floating	S_floating	01	Reserved	Reserved	10	T_floating	T_floating	11	Q_fixed	Reserved																																				
Contents	Meaning for Opcode 16 ₁₆	Meaning for Opcode 14 ₁₆																																																			
00	S_floating	S_floating																																																			
01	Reserved	Reserved																																																			
10	T_floating	T_floating																																																			
11	Q_fixed	Reserved																																																			
8–5	FNC	Instruction class:																																																			
		<table border="0"> <thead> <tr> <th>Contents</th> <th>Meaning for Opcode 16₁₆</th> <th>Meaning for Opcode 14₁₆</th> </tr> </thead> <tbody> <tr> <td>0000</td> <td>ADDx</td> <td>Reserved</td> </tr> <tr> <td>0001</td> <td>SUBx</td> <td>Reserved</td> </tr> <tr> <td>0010</td> <td>MULx</td> <td>Reserved</td> </tr> <tr> <td>0011</td> <td>DIVx</td> <td>Reserved</td> </tr> <tr> <td>0100</td> <td>Reserved</td> <td>ITOFs/ITOfT</td> </tr> <tr> <td>0101</td> <td>CMPxEQ</td> <td>Reserved</td> </tr> <tr> <td>0110</td> <td>CMPxLT</td> <td>Reserved</td> </tr> <tr> <td>0111</td> <td>CMPxLE</td> <td>Reserved</td> </tr> <tr> <td>1000</td> <td>Reserved</td> <td>Reserved</td> </tr> <tr> <td>1001</td> <td>Reserved</td> <td>Reserved</td> </tr> <tr> <td>1010</td> <td>Reserved</td> <td>Reserved</td> </tr> <tr> <td>1011</td> <td>Reserved</td> <td>SQRts/SQRtT</td> </tr> <tr> <td>1100</td> <td>CVTxS</td> <td>Reserved</td> </tr> <tr> <td>1101</td> <td>Reserved</td> <td>Reserved</td> </tr> <tr> <td>1110</td> <td>CVTxT</td> <td>Reserved</td> </tr> <tr> <td>1111</td> <td>CVTxQ</td> <td>Reserved</td> </tr> </tbody> </table>	Contents	Meaning for Opcode 16 ₁₆	Meaning for Opcode 14 ₁₆	0000	ADDx	Reserved	0001	SUBx	Reserved	0010	MULx	Reserved	0011	DIVx	Reserved	0100	Reserved	ITOFs/ITOfT	0101	CMPxEQ	Reserved	0110	CMPxLT	Reserved	0111	CMPxLE	Reserved	1000	Reserved	Reserved	1001	Reserved	Reserved	1010	Reserved	Reserved	1011	Reserved	SQRts/SQRtT	1100	CVTxS	Reserved	1101	Reserved	Reserved	1110	CVTxT	Reserved	1111	CVTxQ	Reserved
Contents	Meaning for Opcode 16 ₁₆	Meaning for Opcode 14 ₁₆																																																			
0000	ADDx	Reserved																																																			
0001	SUBx	Reserved																																																			
0010	MULx	Reserved																																																			
0011	DIVx	Reserved																																																			
0100	Reserved	ITOFs/ITOfT																																																			
0101	CMPxEQ	Reserved																																																			
0110	CMPxLT	Reserved																																																			
0111	CMPxLE	Reserved																																																			
1000	Reserved	Reserved																																																			
1001	Reserved	Reserved																																																			
1010	Reserved	Reserved																																																			
1011	Reserved	SQRts/SQRtT																																																			
1100	CVTxS	Reserved																																																			
1101	Reserved	Reserved																																																			
1110	CVTxT	Reserved																																																			
1111	CVTxQ	Reserved																																																			

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

Table 4–13 VAX Floating-Point Function Field Bit Summary

Bits	Field	Meaning																		
15–13	TRP	Trapping modes: <table><thead><tr><th>Contents</th><th>Meaning for Opcode 14₁₆ and 15₁₆</th></tr></thead><tbody><tr><td>000</td><td>Imprecise (default)</td></tr><tr><td>001</td><td>Underflow enable (/U) – floating-point output Integer overflow enable (/V) – integer output</td></tr><tr><td>010</td><td>UNPREDICTABLE for opcode 15₁₆ instructions Reserved for opcode 14₁₆ instructions</td></tr><tr><td>011</td><td>UNPREDICTABLE for opcode 15₁₆ instructions Reserved for opcode 14₁₆ instructions</td></tr><tr><td>100</td><td>/S – Exception completion enable</td></tr><tr><td>101</td><td>/SU – floating-point output /SV – integer output</td></tr><tr><td>110</td><td>UNPREDICTABLE for opcode 15₁₆ instructions Reserved for opcode 14₁₆ instructions</td></tr><tr><td>111</td><td>UNPREDICTABLE for opcode 15₁₆ instructions Reserved for opcode 14₁₆ instructions</td></tr></tbody></table>	Contents	Meaning for Opcode 14 ₁₆ and 15 ₁₆	000	Imprecise (default)	001	Underflow enable (/U) – floating-point output Integer overflow enable (/V) – integer output	010	UNPREDICTABLE for opcode 15 ₁₆ instructions Reserved for opcode 14 ₁₆ instructions	011	UNPREDICTABLE for opcode 15 ₁₆ instructions Reserved for opcode 14 ₁₆ instructions	100	/S – Exception completion enable	101	/SU – floating-point output /SV – integer output	110	UNPREDICTABLE for opcode 15 ₁₆ instructions Reserved for opcode 14 ₁₆ instructions	111	UNPREDICTABLE for opcode 15 ₁₆ instructions Reserved for opcode 14 ₁₆ instructions
Contents	Meaning for Opcode 14 ₁₆ and 15 ₁₆																			
000	Imprecise (default)																			
001	Underflow enable (/U) – floating-point output Integer overflow enable (/V) – integer output																			
010	UNPREDICTABLE for opcode 15 ₁₆ instructions Reserved for opcode 14 ₁₆ instructions																			
011	UNPREDICTABLE for opcode 15 ₁₆ instructions Reserved for opcode 14 ₁₆ instructions																			
100	/S – Exception completion enable																			
101	/SU – floating-point output /SV – integer output																			
110	UNPREDICTABLE for opcode 15 ₁₆ instructions Reserved for opcode 14 ₁₆ instructions																			
111	UNPREDICTABLE for opcode 15 ₁₆ instructions Reserved for opcode 14 ₁₆ instructions																			

Table 4–13 VAX Floating-Point Function Field Bit Summary (Continued)

Bits	Field	Meaning																																																			
12–11	RND	Rounding modes: <table border="1"> <thead> <tr> <th>Contents</th> <th>Meaning for Opcodes 15₁₆ and 14₁₆</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Chopped (/C)</td> </tr> <tr> <td>01</td> <td>UNPREDICTABLE</td> </tr> <tr> <td>10</td> <td>Normal (default)</td> </tr> <tr> <td>11</td> <td>UNPREDICTABLE</td> </tr> </tbody> </table>	Contents	Meaning for Opcodes 15 ₁₆ and 14 ₁₆	00	Chopped (/C)	01	UNPREDICTABLE	10	Normal (default)	11	UNPREDICTABLE																																									
Contents	Meaning for Opcodes 15 ₁₆ and 14 ₁₆																																																				
00	Chopped (/C)																																																				
01	UNPREDICTABLE																																																				
10	Normal (default)																																																				
11	UNPREDICTABLE																																																				
10–9	SRC	Source datatype: [†] <table border="1"> <thead> <tr> <th>Contents</th> <th>Meaning for Opcode 15₁₆</th> <th>Meaning for Opcode 14₁₆</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>F_floating</td> <td>F_floating</td> </tr> <tr> <td>01</td> <td>D_floating</td> <td>F_floating</td> </tr> <tr> <td>10</td> <td>G_floating</td> <td>G_floating</td> </tr> <tr> <td>11</td> <td>Q_fixed</td> <td>Reserved</td> </tr> </tbody> </table>	Contents	Meaning for Opcode 15 ₁₆	Meaning for Opcode 14 ₁₆	00	F_floating	F_floating	01	D_floating	F_floating	10	G_floating	G_floating	11	Q_fixed	Reserved																																				
Contents	Meaning for Opcode 15 ₁₆	Meaning for Opcode 14 ₁₆																																																			
00	F_floating	F_floating																																																			
01	D_floating	F_floating																																																			
10	G_floating	G_floating																																																			
11	Q_fixed	Reserved																																																			
8–5	FNC	Instruction class: <table border="1"> <thead> <tr> <th>Contents</th> <th>Meaning for Opcode 15₁₆</th> <th>Meaning for Opcode 14₁₆</th> </tr> </thead> <tbody> <tr> <td>0000</td> <td>ADDx</td> <td>Reserved</td> </tr> <tr> <td>0001</td> <td>SUBx</td> <td>Reserved</td> </tr> <tr> <td>0010</td> <td>MULx</td> <td>Reserved</td> </tr> <tr> <td>0011</td> <td>DIVx</td> <td>Reserved</td> </tr> <tr> <td>0100</td> <td>CMPxUN</td> <td>ITOFF</td> </tr> <tr> <td>0101</td> <td>CMPxEQ</td> <td>Reserved</td> </tr> <tr> <td>0110</td> <td>CMPxLT</td> <td>Reserved</td> </tr> <tr> <td>0111</td> <td>CMPxLE</td> <td>Reserved</td> </tr> <tr> <td>1000</td> <td>Reserved</td> <td>Reserved</td> </tr> <tr> <td>1001</td> <td>Reserved</td> <td>Reserved</td> </tr> <tr> <td>1010</td> <td>Reserved</td> <td>SQRTE/SQRTG</td> </tr> <tr> <td>1011</td> <td>Reserved</td> <td>Reserved</td> </tr> <tr> <td>1100</td> <td>CVTxF</td> <td>Reserved</td> </tr> <tr> <td>1101</td> <td>CVTxD</td> <td>Reserved</td> </tr> <tr> <td>1110</td> <td>CVTxG</td> <td>Reserved</td> </tr> <tr> <td>1111</td> <td>CVTxQ</td> <td>Reserved</td> </tr> </tbody> </table>	Contents	Meaning for Opcode 15 ₁₆	Meaning for Opcode 14 ₁₆	0000	ADDx	Reserved	0001	SUBx	Reserved	0010	MULx	Reserved	0011	DIVx	Reserved	0100	CMPxUN	ITOFF	0101	CMPxEQ	Reserved	0110	CMPxLT	Reserved	0111	CMPxLE	Reserved	1000	Reserved	Reserved	1001	Reserved	Reserved	1010	Reserved	SQRTE/SQRTG	1011	Reserved	Reserved	1100	CVTxF	Reserved	1101	CVTxD	Reserved	1110	CVTxG	Reserved	1111	CVTxQ	Reserved
Contents	Meaning for Opcode 15 ₁₆	Meaning for Opcode 14 ₁₆																																																			
0000	ADDx	Reserved																																																			
0001	SUBx	Reserved																																																			
0010	MULx	Reserved																																																			
0011	DIVx	Reserved																																																			
0100	CMPxUN	ITOFF																																																			
0101	CMPxEQ	Reserved																																																			
0110	CMPxLT	Reserved																																																			
0111	CMPxLE	Reserved																																																			
1000	Reserved	Reserved																																																			
1001	Reserved	Reserved																																																			
1010	Reserved	SQRTE/SQRTG																																																			
1011	Reserved	Reserved																																																			
1100	CVTxF	Reserved																																																			
1101	CVTxD	Reserved																																																			
1110	CVTxG	Reserved																																																			
1111	CVTxQ	Reserved																																																			

[†] In the SRC field, both 00 and 01 specify the F_floating source datatype for opcode 14₁₆.

4.7.10 IEEE Standard

The IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754-1985) is included by reference.

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 4.7.7.10, 4.7.7.11, 4.7.8, 4.7.8.3, and Appendix B.

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

Copying 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 4–14.

Table 4–14: Memory Format Floating-Point Instructions Summary

Mnemonic	Operation	Subset
LDF	Load F_floating	VAX
LDG	Load G_floating (Load D_floating)	VAX
LDS	Load S_floating (Load Longword Integer)	Both
LDT	Load T_floating (Load Quadword Integer)	Both
STF	Store F_floating	VAX
STG	Store G_floating (Store D_floating)	VAX
STS	Store S_floating (Store Longword Integer)	Both
STT	Store T_floating (Store Quadword Integer)	Both

4.8.1 Load F_floating

Format:

LDF Fa.wf,disp.ab(Rb.ab) !Memory format

Operation:

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

CASE

 big_endian_data: $va' \leftarrow va \text{ XOR } 100_2$

 little_endian_data: $va' \leftarrow va$

ENDCASE

$Fa \leftarrow (va')\langle 15 \rangle \parallel \text{MAP_F}((va')\langle 14:7 \rangle) \parallel (va')\langle 6:0 \rangle \parallel$
 $(va')\langle 31:16 \rangle \parallel 0\langle 28:0 \rangle$

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, $va\langle 2 \rangle$ (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:

LDG Fa.wg,disp.ab(Rb.ab) !Memory format

Operation:

$$va \leftarrow \{Rbv + \text{SEXT}(\text{disp})\}$$
$$Fa \leftarrow (va)\langle 15:0 \rangle \parallel (va)\langle 31:16 \rangle \parallel (va)\langle 47:32 \rangle \parallel (va)\langle 63:48 \rangle$$

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:

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

CASE

big_endian_data: $va' \leftarrow va \text{ XOR } 100_2$

little_endian_data: $va' \leftarrow va$

ENDCASE

$Fa \leftarrow (va')\langle 31 \rangle \mid \mid \text{MAP}_S((va')\langle 30:23 \rangle) \mid \mid (va')\langle 22:0 \rangle \mid \mid 0\langle 28:0 \rangle$

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, $va\langle 2 \rangle$ (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 $\langle 63:62,58:29 \rangle$, with bits $\langle 61:59 \rangle$ ignored and zeros in bits $\langle 28:0 \rangle$.

An LDS instruction for which the Fa operand is 31 is executed as a PREFETCH_M instruction, described in Section 4.11.8.

4.8.4 Load T_floating

Format:

LDT Fa.wt,disp.ab(Rb.ab) !Memory format

Operation:

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

$Fa \leftarrow (va) \langle 63:0 \rangle$

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.

An LDT instruction for which the Fa operand is 31 is executed as a PREFETCH_MEN instruction, described in Section 4.11.8.

4.8.5 Store F_floating

Format:

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

Operation:

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

CASE

 big_endian_data: $va' \leftarrow va \text{ XOR } 100_2$

 little_endian_data: $va' \leftarrow va$

ENDCASE

$(va')_{<31:0>} \leftarrow Fav_{<44:29>} \mid \mid Fav_{<63:62>} \mid \mid Fav_{<58:45>}$

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:

$$va \leftarrow \{Rbv + \text{SEXT}(\text{disp})\}$$
$$(va)\langle 63:0 \rangle \leftarrow Fav\langle 15:0 \rangle \mid \mid Fav\langle 31:16 \rangle \mid \mid Fav\langle 47:32 \rangle \mid \mid Fav\langle 63:48 \rangle$$

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_floating

Format:

STS Fa.rs,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<63:62> || Fav<58:29>
```

Exceptions:

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

Instruction mnemonics:

STS Store S_floating (Store Longword Integer)

Qualifiers:

None

Description:

STS stores a longword (integer or S_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 S_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.

4.8.8 Store T_floating

Format:

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

Operation:

$$va \leftarrow \{Rbv + \text{SEXT}(\text{disp})\}$$
$$(va)_{<63:0>} \leftarrow Fav_{<63:0>}$$

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 4–15:

Table 4–15: Floating-Point Branch Instructions Summary

Mnemonic	Operation	Subset
FBEQ	Floating Branch Equal	Both
FBGE	Floating Branch Greater Than or Equal	Both
FBGT	Floating Branch Greater Than	Both
FBLE	Floating Branch Less Than or Equal	Both
FBLT	Floating Branch Less Than	Both
FBNE	Floating Branch Not Equal	Both

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 $\pm 1M$ 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, register move, square root, 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 4–16.

Table 4–16 Floating-Point Operate Instructions Summary

Mnemonic	Operation	Subset
Bit and FPCR Operations:		
CPYS	Copy Sign	Both
CPYSE	Copy Sign and Exponent	Both
CPYSN	Copy Sign Negate	Both
CVTLQ	Convert Longword to Quadword	Both
CVTQL	Convert Quadword to Longword	Both
FCMOVxx	Floating Conditional Move	Both
MF_FPCR	Move from Floating-point Control Register	Both
MT_FPCR	Move to Floating-point Control Register	Both
Arithmetic Operations:		
ADDF	Add F_floating	VAX
ADDG	Add G_floating	VAX
ADDS	Add S_floating	IEEE
ADDT	Add T_floating	IEEE

Table 4–16 Floating-Point Operate Instructions Summary (Continued)

Mnemonic	Operation	Subset
Arithmetic Operations, Continued:		
CMPGxx	Compare G_floating	VAX
CMPTxx	Compare T_floating	IEEE
CVTDG	Convert D_floating to G_floating	VAX
CVTGD	Convert G_floating to D_floating	VAX
CVTGF	Convert G_floating to F_floating	VAX
CVTGQ	Convert G_floating to Quadword	VAX
CVTQF	Convert Quadword to F_floating	VAX
CVTQG	Convert Quadword to G_floating	VAX
CVTQS	Convert Quadword to S_floating	IEEE
CVTQT	Convert Quadword to T_floating	IEEE
CVTST	Convert S_floating to T_floating	IEEE
CVTTQ	Convert T_floating to Quadword	IEEE
CVTTS	Convert T_floating to S_floating	IEEE
DIVF	Divide F_floating	VAX
DIVG	Divide G_floating	VAX
DIVS	Divide S_floating	IEEE
DIVT	Divide T_floating	IEEE
FTOIS	Floating-point to integer register move, S_floating	IEEE
FTOIT	Floating-point to integer register move, T_floating	IEEE
ITOFF	Integer to floating-point register move, F_floating	VAX
ITOFS	Integer to floating-point register move, S_floating	IEEE
ITOFT	Integer to floating-point register move, T_floating	IEEE
MULF	Multiply F_floating	VAX
MULG	Multiply G_floating	VAX
MULS	Multiply S_floating	IEEE
MULT	Multiply T_floating	IEEE

Table 4–16 Floating-Point Operate Instructions Summary (Continued)

Mnemonic	Operation	Subset
Arithmetic Operations, Continued:		
SQRTF	Square root F_floating	VAX
SQRTG	Square root G_floating	VAX
SQRTS	Square root S_floating	IEEE
SQRTT	Square root T_floating	IEEE
SUBF	Subtract F_floating	VAX
SUBG	Subtract G_floating	VAX
SUBS	Subtract S_floating	IEEE
SUBT	Subtract T_floating	IEEE

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:

CVTxy Fb.rq,Fc.wx !Floating-point Operate format

Operation:

```
CASE
  CVTQL: Fc ← Fbv<31:30> || 0<2:0> || Fbv<29:0> || 0<28:0>
  CVTLQ: Fc ← SEXT(Fbv<63:62> || Fbv<58:29>)
ENDCASE
```

Exceptions:

Integer Overflow, CVTQL only

Instruction mnemonics:

CVTLQ	Convert Longword to Quadword
CVTQL	Convert Quadword to Longword

Qualifiers:

Trapping:	Exception Completion (/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:

FCMOV_{xx} 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
FCMOVGT	FCMOVE if Register Greater Than Zero
FCMOVLE	FCMOVE if Register Less Than or Equal to Zero
FCMOVL	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:

FCMOV_{xx} 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:

Mx_FPCR Fa.rq,Fa.rq,Fa.wq !Floating-point Operate format

Operation:

```
CASE
  MF_FPCR: Fa ← FPCR
  MT_FPCR: FPCR ← Fav
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

MF_FPCR Move from Floating-point Control Register
MT_FPCR Move to Floating-point Control Register

Qualifiers:

None

Description:

The Floating-point Control Register (FPCR) is read from (MF_FPCR) or written to (MT_FPCR), a floating-point register. The floating-point register to be used is specified by the Fa, Fb, and Fc fields all pointing to the same floating-point register. If the Fa, Fb, and Fc fields do not all point to the same floating-point register, then it is UNPREDICTABLE which register is used. If the Fa, Fb, and Fc fields do not all point to the same floating-point register, the resulting values in the Fc register and in FPCR are UNPREDICTABLE.

If the Fc field is F31 in the case of MT_FPCR, the resulting value in FPCR is UNPREDICTABLE.

The use of these instructions and the FPCR are described in Section 4.7.8.

4.10.5 VAX Floating Add

Format:

ADDx Fa.rx,Fb.rx,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow F_{av} + F_{bv}$

Exceptions:

Invalid Operation
Overflow
Underflow

Instruction mnemonics:

ADDF Add F_floating
ADDG Add G_floating

Qualifiers:

Rounding: Chopped (/C)
Trapping: Exception Completion (/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 4.7.7 for details of the stored result on overflow or underflow.

4.10.6 IEEE Floating Add

Format:

ADDx Fa.rx,Fb.rx,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow F_{a_v} + F_{b_v}$

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: Exception Completion (/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 4.7.7 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 !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:

CMPGEQ	Compare G_floating Equal
CMPGLE	Compare G_floating Less Than or Equal
CMPGLT	Compare G_floating Less Than

Qualifiers:

Trapping: Exception Completion (/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: Exception Completion (/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 may be provided by an operating system (OS) completion handler, because NaNs may 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 exception completion 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:

$F_c \leftarrow \{\text{conversion of } F_b\}$

Exceptions:

Invalid Operation
Integer Overflow

Instruction mnemonics:

CVTGQ Convert G_floating to Quadword

Qualifiers:

Rounding: Chopped (/C)
Trapping: Exception Completion (/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 4.7.7 for details of the stored result on integer overflow.

4.10.10 Convert Integer to VAX Floating

Format:

CVTQy Fb.rq,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow \{\text{conversion of } F_b \langle 63:0 \rangle\}$

Exceptions:

None

Instruction mnemonics:

CVTQF Convert Quadword to F_floating
CVTQG 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:

CVTxy Fb.rx,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow \{\text{conversion of } F_b\}$

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:	Exception Completion (/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 $\text{exp}=0$ 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 4.7.7 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:

CVTTQ Fb.rx,Fc.wq !Floating-point Operate format

Operation:

$F_c \leftarrow \{\text{conversion of } F_b\}$

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: Exception Completion (/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 4.7.7 for details of the stored result on integer overflow and inexact result.

4.10.13 Convert Integer to IEEE Floating

Format:

CVTQy Fb.rq,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow \{\text{conversion of } F_b \text{v} \langle 63:0 \rangle\}$

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: Exception Completion (/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 4.7.7 for details of the stored result on inexact result.

Notes:

- In order to use CVTQS or CVTQT with exception completion 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:

$F_c \leftarrow \{\text{conversion of } F_b\}$

Exceptions:

Invalid Operation

Instruction mnemonics:

CVTST Convert S_floating to T_floating

Qualifiers:

Trapping: Exception Completion (/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:

$F_c \leftarrow \{\text{conversion of } F_{bv}\}$

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: Exception Completion (/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 4.7.7 for details of the stored result on overflow, underflow, or inexact result.

4.10.16 VAX Floating Divide

Format:

DIVx Fa.rx,Fb.rx,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow F_{av} / F_{bv}$

Exceptions:

Invalid Operation
Division by Zero
Overflow
Underflow

Instruction mnemonics:

DIVF Divide F_floating
DIVG Divide G_floating

Qualifiers:

Rounding: Chopped (/C)
Trapping: Exception Completion (/S)
 Underflow Enable (/U)

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 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 $\text{exp}=0$ 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 4.7.7 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:

$F_c \leftarrow F_{a_v} / F_{b_v}$

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: Exception Completion (/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 4.7.7 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 F31.

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.

Implementation Note:

- The FTOIS and FTOIT instructions are supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 1. FTOIS and FTOIT are supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 1. Software emulation of FTOIS and FTOIT is significantly slower than hardware support.

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 R31.

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.

Implementation Note:

- The ITOFF, ITOFS, and ITOFT instructions are supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 1. ITOFF, ITOFS, and ITOFT are supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 1. Software emulation of ITOFF, ITOFS, and ITOFT is significantly slower than hardware support.

4.10.20 VAX Floating Multiply

Format:

MULx Fa.rx,Fb.rx,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow F_{av} * F_{bv}$

Exceptions:

Invalid Operation
Overflow
Underflow

Instruction mnemonics:

MULF Multiply F_floating
MULG Multiply G_floating

Qualifiers:

Rounding: Chopped (/C)
Trapping: Exception Completion (/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 $\text{exp}=0$ 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 4.7.7 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:

$F_c \leftarrow F_{a_v} * F_{b_v}$

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: Exception Completion (/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 4.7.7 for details of the stored result on overflow, underflow, or inexact result.

4.10.22 VAX Floating Square Root

Format:

SQRTx Fb.rx,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow F_b ** (1/2)$

Exceptions:

Invalid operation

Instruction mnemonics:

SQRTF Square root F_floating
SQRTG Square root G_floating

Qualifiers:

Rounding: Chopped (/C)
Trapping: Exception Completion (/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 F31.)

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.

Implementation Notes:

- The SQRTF and SQRTG instructions are supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 1. SQRTF and SQRTG are supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 1. Software emulation of SQRTF and SQRTG is significantly slower than hardware support.

4.10.23 IEEE Floating Square Root

Format:

SQRTx Fb.rx,Fc.wx !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 Enable (/I)
 Exception Completion (/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 F31.) 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.

Implementation Notes:

- The SQRTS and SQRTT instructions are supported in hardware on Alpha implementations for which the AMASK instruction clears feature mask bit 1. SQRTS and SQRTT are supported with software emulation in Alpha implementations for which AMASK does not clear feature mask bit 1. Software emulation of SQRTS and SQRTT is significantly slower than hardware support.

4.10.24 VAX Floating Subtract

Format:

SUBx Fa.rx,Fb.rx,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow F_{av} - F_{bv}$

Exceptions:

Invalid Operation
Overflow
Underflow

Instruction mnemonics:

SUBF Subtract F_floating
SUBG Subtract G_floating

Qualifiers:

Rounding: Chopped (/C)
Trapping: Exception Completion (/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 $\text{exp}=0$ 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 4.7.7 for details of the stored result on overflow or underflow.

4.10.25 IEEE Floating Subtract

Format:

SUBx Fa.rx,Fb.rx,Fc.wx !Floating-point Operate format

Operation:

$F_c \leftarrow F_{av} - F_{bv}$

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

SUBS Subtract S_floating
SUBT Subtract T_floating

Qualifiers:

Rounding: Dynamic (/D)
 Minus infinity (/M)
 Chopped (/C)
Trapping: Exception Completion (/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 4.7.7 for details of the stored result on overflow, underflow, or inexact result.

4.11 Miscellaneous Instructions

Alpha provides the miscellaneous instructions shown in Table 4–17.

Table 4–17: Miscellaneous Instructions Summary

Mnemonic	Operation
AMASK	Architecture Mask
CALL_PAL	Call Privileged Architecture Library Routine
ECB	Evict Cache Block
EXCB	Exception Barrier
FETCH	Prefetch Data
FETCH_M	Prefetch Data, Modify Intent
IMPLVER	Implementation Version
MB	Memory Barrier
PREFETCH	Normal prefetch
PREFETCH_EN	Prefetch Memory Data, Evict Next
PREFETCH_M	Prefetch Memory Data with Modify Intent
PREFETCH_MEN	Prefetch Memory Data with Modify Intent, Evict Next
RPCC	Read Processor Cycle Counter
TRAPB	Trap Barrier
WH64	Write Hint — 64 Bytes
WH64EN	Write Hint — 64 Bytes Evict Next
WMB	Write Memory Barrier

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 21064/21064A/21066/21068/21066A (EV4/EV45/LCA/LCA45 chips), AMASK copies Rbv to Rc.
- On 21164 (EV5), AMASK copies Rbv to Rc.

- On 21164A (EV56), 21164PC (PCA56), 21264/EV6x, and 21364/EV7x, 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.

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

! Mfc 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 4.7.8.1.

4.11.5 Prefetch Data

Format:

FETCHx 0(Rb.ab) !Memory format

Operation:

$va \leftarrow \{Rbv\}$
{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 part of the memory hierarchy that has faster-access, in anticipation of subsequent Load or Store instructions that access that data.

Implementation Note:

FETCHx is intended to help software overlap memory latencies when such latencies are on the order of at least 100 cycles. FETCHx is unlikely to help (or be implemented) for significantly shorter memory latencies. Code scheduling and cache-line prefetching (see Section A.3.6) should be used to overlap such shorter latencies.

Existing Alpha implementations (through the 21364) have memory latencies that are too short to profitably implement FETCHx. Therefore, FETCHx does not improve memory performance in existing Alpha implementations.

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.

4.11.6 Implementation Version

Format:

IMPLVER Rc !Operate format

Operation:

Rc ← value, which is defined in Appendix D

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.

Notes:

- 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.

4.11.7 Memory Barrier

Format:

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:

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 Section 5.6 for details on using the MB instruction to serialize these accesses. Section 5.6 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 Prefetch Memory Data

Format:

PREFETCH_x disp.ab(Rb.ab) !Memory format

Operation:

```
CASE
  PREFETCH:      LDL  R31, disp (Rb)
  PREFETCH_EN:   LDQ  R31, disp (Rb)
  PREFETCH_M:    LDS  F31, disp (Rb)
  PREFETCH_MEN:  LDT  F31, disp (Rb)
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

PREFETCH	Normal Prefetch
PREFETCH_EN	Prefetch, Evict Next
PREFETCH_M	Prefetch with Modify Intent
PREFETCH_MEN	Prefetch with Modify Intent, Evict Next

Qualifiers:

None

Description:

A prefetch is a hint to the processor that a cache block might be used in the future and should be brought into the cache now.

A prefetch with modify intent is a hint to the processor that a cache block might be modified in the future and should be brought into the cache now with write permission.

A prefetch, evict next, is a hint to the processor that a cache block should be brought into the cache now and marked for preferential eviction on future cache fills. Such a prefetch is particularly useful with an associative cache, to prefetch data that is not repeatedly referenced — data that has a short temporal lifetime in the cache. If such a cache block might require write permission, the prefetch is also specified with modify intent.

The PREFETCHx instructions perform different types of cache block prefetches, as follows:

Instruction	Operation
PREFETCH	If possible, the addressed cache block is allocated to the Dcache with read permission.
PREFETCH_EN	Prefetch the addressed cache block and mark it for preferential eviction on future cache fills.
PREFETCH_M	If possible, the addressed cache block is allocated to the Dcache with write permission.
PREFETCH_MEN	Prefetch the addressed cache block with modify intent and mark it for preferential eviction on future cache fills.

Implementation Notes:

- PREFETCH and PREFETCH_EN only affect performance and do not modify any architecturally visible state.
- PREFETCH_M and PREFETCH_MEN only affect performance except for possibly signalling a floating-point disabled exception or for their effects on LDx_L/STx_C sequences.
- PREFETCH_M and PREFETCH_MEN must not trap on processors that choose not to implement floating-point support. On processors that do implement floating-point support, it is UNPREDICTABLE whether PREFETCH_M and PREFETCH_MEN can generate a floating-point disable exception.
- Eviction policy is implementation-dependent and is described in the hardware reference manual for the particular implementation. Consult Chapter 2 in the appropriate manual, available at ftp.compaq.com/pub/products/alphaCPUdocs.

4.11.9 Read Processor Cycle Counter

Format:

RPCC Ra.wq, Rb.rq !Memory format

Operation:

{see programming note for use of Rb}
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.

The RPCC instruction is not issued until all previous instructions that generate a result in Rb have completed.

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 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
```

RPCC does not read the Processor Cycle Counter (PCC) any earlier than the generation of a result by the nearest preceding instruction that modifies register Rb. If R31 is used as the Rb operand, the PCC need not wait for any preceding computation.

Programming Note

See Section E.1.4 for information about RPCC and various Alpha processor implementations.

4.11.10 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 exception 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.

4.11.11 Write Hint

Format:

WH64x (Rb.ab) ! Mfc 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
WH64EN	Write Hint - 64 Bytes Evict Next

Qualifiers:

None

Description:

The WH64x 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 1.6.2.

The WH64x 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, WH64x is treated as a NOP. If WH64x is not supported on a particular Alpha implementation, it is treated as a NOP.

WH64EN is a hint to the processor that the corresponding 64-byte cache block should have a short temporal lifetime in the cache and can be marked for preferential eviction in future cache fills.

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 WH64x instruction followed by eight quadword stores for each aligned 64-byte region to be written.

Implementation Notes:

- 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 WH64x as defined. However, they may instead implement the instruction by allocating a smaller aligned cache block for write access or by treating WH64x as a NOP.
- Processors with cache blocks larger than 64 bytes are also encouraged to implement WH64x as defined. However, they may instead treat WH64x as a NOP.
- WH64EN is implemented as a NOP on processors previous to the 21264/EV6x and implemented as WH64 on 21264/EV6x processors.
- WH64 and WH64EN differ only in their eviction policy, and that policy is implementation-dependent. The eviction policy for particular implementations is described in the appropriate hardware reference manual, which can be found at ftp.com-paq.com/pub/products/alphaCPUdocs.

4.11.12 Write Memory Barrier

Format:

WMB

!Memory format

Operation:

```
{ Guarantee that
{ All preceding stores that access memory-like
{   regions are ordered before any subsequent stores
{   that access memory-like regions and
{ All preceding stores that access non-memory-like
{   regions are ordered before any subsequent stores
{   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:

Processor i	Processor j
<Acquire lock>	
MB	
<Read and write data in shared section>	
WMB	
<Release lock>	⇒ <Acquire lock>
	MB
	<Read and write data in shared section>
	WMB

The example above is similar to that in Section 5.5.5, 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:

1. 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).
2. For each read u of shared data in the critical section, there is a write v such that:
 - a. v is BEFORE the WMB.
 - b. v follows u in processor issue sequence (see Section 5.6.1.1).
 - c. v either depends on u (see Section 5.6.1.7) or overlaps u (see Section 5.6.1), or both.
3. 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 4–18 for use in translated VAX code. These instructions 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 4–18: VAX Compatibility Instructions Summary

Mnemonic	Operation
RC	Read and Clear
RS	Read and Set

4.13 Multimedia (Graphics and Video) Support

Alpha provides the following instructions that enhance support for graphics and video algorithms:

Mnemonic	Operation
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
PERR	Pixel Error
PKLB	Pack Longwords to Bytes
PKWB	Pack Words to Bytes
UNPKBL	Unpack Bytes to Longwords
UNPKBW	Unpack Bytes to Words

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.

Implementation Note:

Alpha processors for which the AMASK instruction clears feature mask bit 8 implement these instructions. Those processors for which AMASK does not clear feature mask bit 8 can take an Illegal Instruction trap, and software can emulate their function, if required.

4.13.1 Byte and Word Minimum and Maximum

Format:

MINxxx	Ra.rq,Rb.rqRc.wq Ra.rq,#b.ib,Rc.wq	! Operate Format
MAXxxx	Ra.rq,Rb.rq,Rc.wq Ra.rq,#b.ib,Rc.wq	! Operate Format

Operation:

```
CASE
  MINUB8:
    FOR i FROM 0 TO 7
      Rcv<i*8+7:i*8> = MINU(Rav<i*8+7:i*8>,Rbv<i*8+7:i*8>)
    END
  MINSB8:
    FOR i FROM 0 TO 7
      Rcv<i*8+7:i*8> = MINS(Rav<i*8+7:i*8>,Rbv<i*8+7:i*8>)
    END
  MINUW4:
    FOR i FROM 0 TO 3
      Rcv<i*16+15:i*16> = MINU(Rav<i*16+15:i*16>,Rbv<i*16+15:i*16>)
    END
  MINSW4:
    FOR i FROM 0 TO 3
      Rcv<i*16+15:i*16> = MINS(Rav<i*16+15:i*16>,Rbv<i*16+15:i*16>)
    END
  MAXUB8:
    FOR i FROM 0 TO 7
      Rcv<i*8+7:i*8> = MAXU(Rav<i*8+7:i*8>,Rbv<i*8+7:i*8>)
    END
  MAXSB8:
    FOR i FROM 0 TO 7
      Rcv<i*8+7:i*8> = MAXS(Rav<i*8+7:i*8>,Rbv<i*8+7:i*8>)
    END
  MAXUW4:
    FOR i FROM 0 TO 3
      Rcv<i*16+15:i*16> = MAXU(Rav<i*16+15:i*16>,Rbv<i*16+15:i*16>)
    END
  MAXSW4:
    FOR i FROM 0 TO 3
      Rcv<i*16+15:i*16> = MAXS(Rav<i*16+15:i*16>,Rbv<i*16+15:i*16>)
    END
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
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

Rb.rq,Rc.wq

! Operate Format

Operation:

```
CASE
  PKLB:
    BEGIN
      Rc<07:00> ← Rbv<07:00>
      Rc<15:08> ← Rbv<39:32>
      Rc<63:16> ← 0
    END
  PKWB:
    BEGIN
      Rc<07:00> ← Rbv<07:00>
      Rc<15:08> ← Rbv<23:16>
      Rc<23:16> ← Rbv<39:32>
      Rc<31:24> ← Rbv<55:48>
      Rc<63:32> ← 0
    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:

```
temp = 0
CASE
  UNPKBL:
    BEGIN
      temp<07:00> = Rbv<07:00>
      temp<39:32> = Rbv<15:08>
    END
  UNPKBW:
    BEGIN
      temp<07:00> = Rbv<07:00>
      temp<23:16> = Rbv<15:08>
      temp<39:32> = Rbv<23:16>
      temp<55:48> = Rbv<31:24>
    END
ENDCASE
Rc ← temp
```

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.

System Architecture and Programming Implications (I)

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
- Translation buffers and virtual caches
- Caches and write buffers
- 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:

- 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).
- 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.
- 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 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.

- 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.)
- Write-back caches must be able to detect a later write from another processor and invalidate or update the cache contents.
- A processor must guarantee that a data store to a location followed by a data load from the same location reads the updated value.
- 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.
- 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.
- 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.
- 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 either:

- any Store to the same physical address that is subsequently fetched as an instruction by some corresponding (virtual address, ASN) pair, or
- any change to 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 by 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.
- 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 LD_x_L/ST_x_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:

```
try_again:
    LDQ_L  R1,x
    <modify R1>
    STQ_C  R1,x
    BEQ   R1,no_store
    :
no_store:
    <code to check for excessive iterations>
    BR    try_again
```

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 by 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 LD_x_L and ST_x_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:

- 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 complete quickly and successfully.
- 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.
- Only the fall-through path of the `BLBS` instructions does a `STx_C`; some implementations may not allow a successful `STx_C` after a branch-taken.
- Only register-to-register operate instructions are used to do the modify.
- 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.

- 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.
- The memory barrier instructions are discussed in Section 5.5.5. 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.

- 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 Prefetching Low-Contention Atomic Data and Locks

A low-contention situation is one in which multiple processors are not vigorously contending for the same datum. In a low-contention situation, performance can be improved by executing a prefetch-with-modify-intent well in advance of attempting an atomic update or of attempting to set an atomic lock.

```
LDA      R3,0x1000      # test AMASK<12>. See Section E.1.6.
AMASK   R3,R3
```

```

        BNE      R3, skip_prefetch

        LDS      F31, 0(R1)      # Prefetch with modify intent (PREFETCH_M)
                                # to prefetch the cache block
                                # with the lock in it exactly
                                # once per lock acquisition.

skip_prefetch:
        .          # 20 to 80 cycles ahead of the
        .          # atomic memory ref to overcome
        .          # memory latency if possible.
start:
        LDA      R2, 1(R31)
        LDQ_L    R0, 0(R1)
        BNE      R0, lazy
        STQ_C    R2, 0(R1)
        BEQ      R2, start
        BR       done
lazy:
        LDQ      R0, 0(R1)
        BNE      R0, lazy
        BR       start
done:

```

Notice that this code does not use the spin-loop, shown in the example code in Section 5.5.3, which is suitable only for high-contention locks. Notice also relative to the code in Section 5.5.2, the prefetch is executed before the atomic update.

The code above can be particularly useful in large multiprocessor systems with significant latencies. With this code, only one system transaction is required for the lock to succeed because the cache block that contains the lock is brought into the cache with write permission. Without the prefetch-with-modify-intent, two system transactions can be required: one for the LDx_L to read the block into the cache and one for the STx_C to get permission to write the block.

Note:

When a prefetch-with-modify-intent issues a system transaction to get write permission (or ownership) of the block, the prefetch is issuing a transaction similar to a store. And, like a store, such a prefetch can clear the lock flag on another processor.

5.5.5 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>
<endcode_example>
```

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:

1. 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).
2. For each read u of shared data in the critical section, there is a write v such that:
 - a. v is BEFORE the WMB.
 - b. v follows u in processor issue sequence (see Section 5.6.1.1).
 - c. v either depends on u (see Section 5.6.1.7) or overlaps u (see Section 5.6.1), or both.
3. 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 architecture, 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 $Pi:Op<m>(x,a)$, 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 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 $Pi:Op<m>(x,a)$, where the address x is divisible by size m .

The word "access" is also used as a verb; a read/write access $Pi:Op<m>(x,a)$ accesses byte z if $x \leq z < x+m$. Two read/write accesses $Op1<m>(x,a)$ and $Op2<n>(y,b)$ 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 (\Leftarrow) 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 (\Leftarrow), 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 $u \Leftarrow v$, 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, as shown in Table 5–1.

Table 5–1 Processor Issue Constraints

1st↓ 2nd →	Pi:I<n=4>(y,b)	Pi:R<n>(y,b)	Pi:W<n>(y,b)	Pi:MB	Pi:IMB
Pi:I<m=4>(x,a)	\Leftarrow if overlap		\Leftarrow if overlap	\Leftarrow	\Leftarrow
Pi:R<m>(x,a)		\Leftarrow if overlap	\Leftarrow if overlap	\Leftarrow	\Leftarrow
Pi:W<m>(x,a)			\Leftarrow if overlap	\Leftarrow	\Leftarrow
Pi:MB		\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow
Pi:IMB	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow	\Leftarrow

Where "overlap" denotes the condition $\max(x,y) < \min(x+m,y+n)$.

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.
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 non-overlapping 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 $P_i:R\langle n \rangle(y,b)$ to be ordered BEFORE an overlapping write access $P_i:W\langle m \rangle(x,a)$ 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 $P_i:I\langle 4 \rangle(y,b)$ may also be ordered BEFORE an overlapping write $P_i:W\langle m \rangle(x,a)$ 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 (\Leftarrow) ordering, that is, either $u \Leftarrow v$ or $v \Leftarrow u$.

There is no direct requirement that nonoverlapping accesses be comparable in the BEFORE (\Leftarrow) 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 $P_i:W\langle m \rangle(x,a)$ and v is an overlapping read access $P_j:R\langle n \rangle(y,b)$, u is visible to v only if:

$u \Leftarrow v$, or

u precedes v in processor issue sequence (possible only if $P_i=P_j$).

If u is a write access $P_i:W\langle m \rangle(x,a)$ and v is an overlapping instruction fetch $P_j:I\langle 4 \rangle(y,b)$, there are the following rules for visibility:

1. If $u \Leftarrow v$, 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 overlaps w and precedes w in processor issue sequence, and
 - w is visible to v ,
 - then u is visible to v .
 - b. If there is an instruction fetch w such that:
 - u is visible to w , and
 - w overlaps v and precedes v 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 $P_i:W\langle m \rangle(x,a)$, and v is either $P_j:I\langle 4 \rangle(y,b)$ or $P_j:R\langle n \rangle(y,b)$, 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 . In this situation, u is a source of v .

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 P_i to P_j is modeled as P_i writing to a location INT_{ij} , and P_j reading from INT_{ij} .)

5.6.1.7 Definition of Dependence Constraint

The depends relation (DP) is defined as follows. Given u and v issued by processor P_i , 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:

- u determines the execution of v , the location accessed by v , or the value written by v .
- 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).

Note that the DP relation does not directly impose a BEFORE (\Leftarrow) ordering between accesses u and v .

The dependence constraint requires that the union of the DP relation and the "is a source of" relation (see Section 5.6.1.6) be acyclic. That is, there must not exist reads and/or I-fetches R_1, \dots, R_n , and writes W_1, \dots, W_n , such that:

1. $n \geq 1$,
2. For each i , $1 \leq i \leq n$, R_i DP W_i ,
3. For each i , $1 \leq i < n$, W_i is a source of R_{i+1} , and
4. W_n is a source of R_1 .

That constraint eliminates the possibility of "causal loops." A simple example of a "causal loop" is when the execution of a write on P_i depends on the execution of a write on P_j 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 P_i executes:

```
LDQ   R1, x
STQ   R1, y
```

Processor P_j 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:

P _i	P _j
[U1] P _i :R<8>(x,0)	[V1] P _j :R<8>(y,0)
[U2] P _i :W<8>(y,0)	[V2] P _j :W<8>(x,0)

Analysis:

- <1> By the definitions of storage and visibility, U₂ is the source of V₁, and V₂ is the source of U₁.
- <2> By the definition of DP and examination of the code, U₁ DP U₂, and V₁ DP V₂.
- <3> Thus, U₁ DP U₂, U₂ is the source of V₁, V₁ DP V₂, and V₂ is the source of U₁. This circular chain 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 P_i's program:

```
LDQ   R1, x
BNE   R1, done
STQ   R31, y
done:
```

Processor P_j'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 $w \Leftarrow u$ or $v \Leftarrow w$.

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 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)

Initially, location x contains 1:

Pi	Pj
[U1]Pi:W<8>(x,2)	[V1]Pj:R<8>(x,2)
	[V2]Pj:R<8>(x,1)

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 $U1 \Leftarrow V1$.
- <3> By the processor issue constraints (Section 5.6.1.3), $V1 \Leftarrow V2$.
- <4> By the transitivity of the partial order \Leftarrow , it follows from <2> and <3> that $U1 \Leftarrow V2$.
- <5> By the rules for visibility, it follows from $U1 \Leftarrow V2$ that U1 is visible to V2.
- <6> Since U1 is AFTER the initialization of x , U1 is the latest (in the \Leftarrow 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)

Initially, location x contains 1:

Pi	Pj
[U1]Pi:W<8>(x,2)	[V1]Pj:W<8>(x,3)
	[V2]Pj:R<8>(x,2)
	[V3]Pj:R<8>(x,3)

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 \Leftarrow order) write to x visible to V2.
- <3> From <1> and <2>, $V1 \Leftarrow U1$.
- <4> Since U1 is visible to V2, and they are issued by different processors, $U1 \Leftarrow V2$.
- <5> By the processor issue constraints, $V2 \Leftarrow V3$.
- <6> From <4> and <5>, $U1 \Leftarrow V3$.
- <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)

Initially, location x contains 1:

Pi	Pj	Pk
[U1]Pi:W<8>(x,2)	[V1]Pj:W<8>(x,3)	[W1]Pk:R<8>(x,3)
[U2]Pi:R<8>(x,3)		[W2]Pk:R<8>(x,2)

Analysis:

- <1> U2 reading 3 implies V1 is the latest write to x visible to U2, therefore $U1 \Leftarrow V1$.
- <2> W1 reading 3 implies V1 is visible to W1, so $V1 \Leftarrow W1 \Leftarrow W$, therefore V1 is also visible to W2.
- <3> W2 reading 2 implies U1 is the latest write to x visible to W2, therefore $V1 \Leftarrow U1$.
- <4> From <1> and <3>, $U1 \Leftarrow V1 \Leftarrow U1$.

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)

Initially, locations x and y contain 1:

Pi	Pj
[U1]Pi:W<8>(x,2)	[V1]Pj:R<8>(y,2)
[U2]Pi:W<8>(y,2)	[V2]Pj:R<8>(x,1)

Analysis:

- <1> V1 reading 2 implies $U2 \Leftarrow V1$, by storage and visibility.
- <2> Since V2 does not read 2, there cannot be $U1 \Leftarrow V2$.
- <3> By the access order constraints, it follows from <2> that $V2 \Leftarrow U1$.

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)

Initially, locations x and y contain 1:

Pi	Pj
[U1]Pi:W<8>(x,2)	[V1]Pj:R<8>(y,2)
	[V2]Pj:MB
[U2]Pi:W<8>(y,2)	[V3]Pj:R<8>(x,1)

Analysis:

- <1> V1 reading 2 implies $U2 \Leftarrow V1$, by storage and visibility.
- <2> $V1 \Leftarrow V2 \Leftarrow V3$, by processor issue constraints.
- <3> V3 reading 1 implies $V3 \Leftarrow U1$, by storage and visibility.

There is $U2 \Leftarrow V1 \Leftarrow V2 \Leftarrow V3 \Leftarrow U1$. 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)

Initially, locations x and y contain 1:

Pi	Pj
[U1]Pi:W<8>(x,2)	[V1]Pj:R<8>(y,2)
[U2]Pi:MB	
[U3]Pi:W<8>(y,2)	[V2]Pj:R<8>(x,1)

Analysis:

- <1> $U1 \Leftarrow U2 \Leftarrow U3$, by processor issue constraints.
- <2> $V1$ reading 2 implies $U3 \Leftarrow V1$, by storage and visibility.
- <3> $V2$ reading 1 implies $V2 \Leftarrow U1$, by storage and visibility.

There is $V2 \Leftarrow U1 \Leftarrow U2 \Leftarrow U3 \Leftarrow V1$. 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)

Initially, locations x and y contain 1:

Pi	Pj
[U1]Pi:W<8>(x,2)	[V1]Pj:R<8>(y,2)
[U2]Pi:MB	[V2]Pj:MB
[U3]Pi:W<8>(y,2)	[V3]Pj:R<8>(x,1)

Analysis:

- <1> $V3$ reading 1 implies $V3 \Leftarrow U1$, by storage and visibility.
- <2> $V1$ reading 2 implies $U3 \Leftarrow V1$, by storage and visibility.
- <3> $U1 \Leftarrow U2 \Leftarrow U3$, by processor issue constraints.
- <4> $V1 \Leftarrow V2 \Leftarrow V3$, by processor issue constraints.
- <5> By <2>, <3>, and <4>, $U1 \Leftarrow U2 \Leftarrow U3 \Leftarrow V1 \Leftarrow V2 \Leftarrow V3$.

Both <1> and <5> cannot be true, so if $V1$ reads 2, then $V3$ must also read 2.

If both x and y are in memory-like regions, the sequence remains impossible if $U2$ is changed to a WMB. Similarly, if both x and y are in non-memory-like regions, the sequence remains impossible if $U2$ is changed to a WMB.

5.6.2.8 Litmus Test 8 (Impossible Sequence)

Initially, locations x and y contain 1:

Pi	Pj
[U1]Pi:W<8>(x,2)	[V1]Pj:W<8>(y,2)
[U2]Pi:MB	[V2]Pj:MB
[U3]Pi:R<8>(y,1)	[V3]Pj:R<8>(x,1)

Analysis:

- <1> V3 reading 1 implies $V3 \Leftarrow U1$, by storage and visibility.
- <2> U3 reading 1 implies $U3 \Leftarrow V1$, by storage and visibility.
- <3> $U1 \Leftarrow U2 \Leftarrow U3$, by processor issue constraints.
- <4> $V1 \Leftarrow V2 \Leftarrow V3$, by processor issue constraints.
- <5> By <2>, <3>, and <4>, $U1 \Leftarrow U2 \Leftarrow U3 \Leftarrow V1 \Leftarrow V2 \Leftarrow V3$.

Both <1> and <5> cannot be true, so if U3 reads 1, then V3 must read 2, and vice versa.

5.6.2.9 Litmus Test 9 (Impossible Sequence)

Initially, location x contains 1:

Pi	Pj
[U1]Pi:W<8>(x,2)	[V1]Pj:W<8>(x,3)
[U2]Pi:R<8>(x,2)	[V2]Pj:R<8>(x,3)
[U3]Pi:R<8>(x,3)	[V3]Pj:R<8>(x,2)

Analysis:

- <1> V3 reading 2 implies U1 is the latest write to x visible to V3, therefore $V1 \Leftarrow U1$.
- <2> U3 reading 3 implies V1 is the latest write to x visible to U3, therefore $U1 \Leftarrow V1$.

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 100000001_{16} :

Pi	Pj
[U1]Pi:W<4>(x,2)	[V1]Pj:W<4>(x+4,2)
[U2]Pi:R<8>(x,100000002 ₁₆)	[V2]Pj:R<8>(x,200000001 ₁₆)

Analysis:

- <1> Since U2 reads 1 from $x+4$, V1 is not visible to U2. Thus $U2 \Leftarrow V1$.
- <2> Similarly, $V2 \Leftarrow U1$.
- <3> U1 is visible to U2, but since they are issued by the same processor, it is not necessarily the case that $U1 \Leftarrow U2$.
- <4> Similarly, it is not necessarily the case that $V1 \Leftarrow V2$.

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 100000001_{16} :

Pi	Pj
[U1]Pi:W<4>(x,2)	[V1]Pj:R<8>(x,200000001 ₁₆)
[U2]Pi:MB or WMB	
[U3]Pi:W<4>(x+4,2)	

Analysis:

- <1> V1 reading 200000001_{16} implies $U3 \Leftarrow V1 \Leftarrow U1$ by storage and visibility.
- <2> $U1 \Leftarrow U2 \Leftarrow U3$, 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 PALcode
- 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 PALcode/cache implementations may have an implied CALL_PAL IMB in the I-stream TB fill routine, but this is transparent to the non-PALcode 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 Multiprocessor 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 equivalent¹ 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 MB¹ 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.

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.

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 unless the PTE is marked to indicate no such ordering is required.

5.6.4.4 Multiprocessor 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.

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 MB¹ 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 Multiprocessor 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.

¹ See Footnote on page 5-23.

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:

First Processor	Second Processor
:	
Save state of current process.	
MB [1]	
Pass ownership of process context data structure memory.	⇒ Pick up ownership of process context data structure memory.
	MB [2]
	Restore state of new process context data structure memory.
	Make I-stream coherent [3].
	Make TB coherent [4].
	:
	Execute code for new process that accesses memory that is not common to all processes.

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 the following, where, on a single processor, there is no need for the barriers:

First Processor	Second Processor
:	
Pick up ownership of process context data structure memory.	
MB	
Assign new ASN or invalidate TBs.	
Save state of current process.	
Restore state of new process.	
MB	
Pass ownership of process context data structure memory.	⇒
:	Pick up ownership of new process context data structure memory.
:	MB
	Assign new ASN or invalidate TBs.
	Save state of current process.
	Restore state of new process.
	MB
	Pass ownership of old process context data structure memory.
	:
	Execute code for new process that accesses memory that is not common to all processes.

5.6.4.6 Multiprocessor 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:

First Processor	Second Processor
:	
Write data	
MB	

First Processor	⇒	Second Processor
Send interrupt		Receive interrupt
		MB
		Access data
		:

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 MB¹.
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.

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.

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 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 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:

First Processor	Second Processor
:	
Write CSR_A	
MB	
Write flag (in memory)	⇒ Read flag (in memory)
	MB
	Write CSR_B
	:

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. 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 can follow the rules in Section 4.7.7.3 to reliably bound how far the PC may advance before taking a trap, how far an UNPREDICTABLE result may propagate or continue from a trap by supplying a well-defined result R within an arithmetic trap handler. Arithmetic instructions that do not use the /S exception completion qualifier can reliably produce that behavior by inserting TRAPB instructions at appropriate points.

Common PALcode Architecture (I)

6.1 PALcode

In a family of machines, both users and operating system developers 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.

An Alpha Privileged Architecture Library (PALcode) of routines and environments is supplied by Compaq. Other systems may use a library supplied by Compaq or architect and implement a different library of routines. Alpha systems are required to support the replacement of PALcode defined by Compaq 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 Compaq with an operating system-specific version. The following functions must be implemented in PALcode, *not* directly in hardware, to facilitate replacement with different versions.

- 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.
- 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. Compaq defines the operation of these PALcode instructions for operating system implementations supplied by Compaq.

Table 6–1: PALcode Instructions that Require Recognition

OpenVMS Mnemonic	Tru64 UNIX and Alpha Linux Mnemonic	Operation
BPT	bpt	Breakpoint trap
BUGCHK	bugchk	Bugcheck trap
CSERVE	cserve	Console service
GENTRAP	gentrap	Generate trap
READ_UNQ	rdunique	Read unique value
SWPPAL	swppal	Swap PALcode
WRITE_UNQ	wrunique	Write unique value

The PALcode 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

OpenVMS Mnemonic	Tru64 UNIX and Alpha Linux Mnemonic	Type	Operation
DRAINA	draina	Privileged	Drain aborts
HALT	halt	Privileged	Halt processor
IMB	imb	Unprivileged	I-stream memory barrier

6.7.1 Drain Aborts

Format:

CALL_PAL DRAINA !PALcode format

Operation:

```
IF PS<literal>(<)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 Drain Aborts

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, DRAINA 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 !PALcode format

Operation:

```
IF PS<literal><>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 Section 27.5.4.

6.7.3 Instruction Memory Barrier

Format:

CALL_PAL IMB !PALcode format

Operation:

```
{Make instruction stream coherent with data stream}
IF PS<CM> NE 0
    IF {Tru64 UNIX and Alpha Linux PALcode}
        (PCBB+40)<32> <- 1
    IF {OpenVMS PALcode}
        (PCBB+56)<32> <- 1
```

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. When an IMB is executed in other than kernel mode, that fact is recorded in the operating system HWPCB (or PCB) at HWPCB<IMB> to help software manage those multiprocessor events. Software is responsible for clearing the HWPCB<IMB> bit, as appropriate.

The cache coherency and sharing rules are described in Section 5.4.

Chapter 7

Console Subsystem Overview (I)

On an Alpha system, underlying control of the system platform hardware is provided by a *console subsystem*. The console subsystem:

- Initializes, tests, and prepares the system platform hardware for Alpha system software.
- Bootstraps (loads into memory and starts the execution of) system software.
- Controls and monitors the state and state transitions of each processor in a multiprocessor system.
- Provides services to system software that simplify system software control of and access to platform hardware.
- 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.

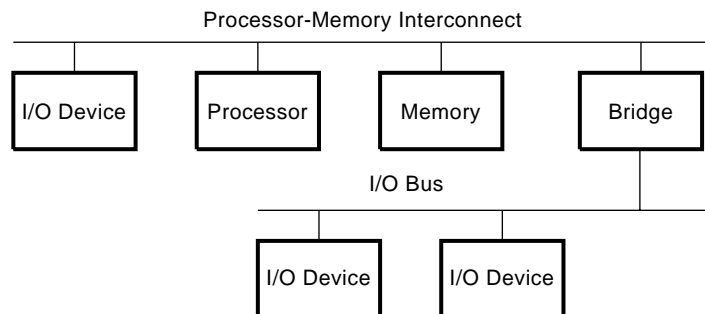
The console subsystem interacts with the console operator through a virtual display device or *console terminal*. The console operator may be a person or a management application.

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

OpenVMS Software (II-A)

The following chapters describe how the OpenVMS operating system relates to the Alpha architecture:

- Chapter 9, Introduction to OpenVMS (II-A)
- Chapter 10, PALcode Instruction Descriptions (II-A)
- Chapter 11, Memory Management (II-A)
- Chapter 12, Process Structure (II-A)
- Chapter 13, Internal Processor Registers (II-A)
- Chapter 14, Exceptions, Interrupts, and Machine Checks (II-A)

Introduction to OpenVMS (II–A)

The goals of this design are to provide a hardware-implementation independent interface between the OpenVMS operating system and the hardware. Further, the design provides the needed abstractions to minimize the impact between OpenVMS and different hardware implementations. Finally, the design must contain only that overhead necessary to satisfy those requirements, while still supporting high-performance systems.

9.1 Register Usage

In addition to those registers described in Chapter 3, OpenVMS defines the registers described in the following sections.

9.1.1 Processor Status

The Processor Status (PS) is a special register that contains the current status of the processor. It can be read by the `CALL_PAL RD_PS` instruction. The software field `PS<SW>` can be written by the `CALL_PAL WR_PS_SW` routine. See Section 14.2.1 for a description of the PS register.

9.1.2 Stack Pointer (SP)

Integer register R30 is the Stack Pointer (SP).

The SP contains the address of the top of the stack in the current mode.

Certain PALcode instructions, such as `CALL_PAL REI`, use R30 as an implicit operand. During such operations, the address value in R30, interpreted as an unsigned 64-bit integer, decreases (predecrements) when items are pushed onto the stack and increases (postincrements) when they are popped from the stack. After pushing (writing) an item to the stack, SP points to that item.

9.1.3 Internal Processor Registers (IPRs)

The IPRs provide an architected mapping to internal hardware or provide other specialized uses. They are available only to privileged software through PALcode routines and allow OpenVMS to interrogate or modify system state. The IPRs are described in Chapter 13.

9.1.4 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>) are an offset, PCC_OFF. PCC_OFF is a value that, when added to PCC_CNT, gives the total PCC register count for this process, modulo $2^{**}32$.

Chapter 10

PALcode Instruction Descriptions (II–A)

This chapter describes the PALcode instructions that are implemented for the OpenVMS environment. The PALcode instructions are a set of unprivileged and privileged CALL_PAL instructions that are used to match specific operating system requirements to the underlying hardware implementation.

For example, privileged PALcode instructions switch the hardware context of a process structure. Unprivileged PALcode instructions implement the uninterruptible queue operations. Also, PALcode instructions provide mechanisms for standard interrupt and exception reporting that are independent of the underlying hardware implementation.

Table 10–1 lists all the unprivileged and privileged OpenVMS PALcode instructions and the section in which they are described.

Table 10–1: OpenVMS PALcode Instructions

Mnemonic	Operation	Section
AMOVRM	Atomic move register/memory	10.4.1
AMOVRR	Atomic move register/register	10.4.1
BPT	Breakpoint	10.1.1
BUGCHK	Bugcheck	10.1.2
CFLUSH	Cache flush	10.6.1
CHME	Change mode to executive	10.1.3
CHMK	Change mode to kernel	10.1.4
CHMS	Change mode to supervisor	10.1.5
CHMU	Change mode to user	10.1.6
CLRFEN	Clear floating-point enable	10.1.7
CSERVE	Console service	10.6.2
DRAINA	Drain aborts	6.7.1
GENTRAP	Generate software trap	10.1.8
HALT	Halt processor	6.7.2
IMB	I-stream memory barrier	6.7.3
INSQxxx	Insert in specified queue	10.3

Table 10–1: OpenVMS PALcode Instructions (Continued)

Mnemonic	Operation	Section
LDQP	Load quadword physical	10.6.3
MFPR	Move from processor register	10.6.4
MTPR	Move to processor register	10.6.5
PROBER	Probe read access	10.1.9
PROBEW	Probe write access	10.1.9
RD_PS	Read processor status	10.1.10
READ_UNQ	Read unique context	10.5.1
REI	Return from exception or interrupt	10.1.11
REMQxxx	Remove from specified queue	10.3
RSCC	Read system cycle counter	10.1.12
STQP	Store quadword physical	10.6.6
SWASTEN	Swap AST enable	10.1.13
SWPCTX	Swap privileged context	10.6.7
SWPPAL	Swap PALcode image	10.6.8
WRITE_UNQ	Write unique context	10.5.2
WR_PS_SW	Write processor status software field	10.1.14
WTINT	Wait for interrupt	10.6.9

10.1 Unprivileged General PALcode Instructions

The general unprivileged instructions in this section, together with those in Sections 10.3, 10.4, and 10.5, provide support for the underlying OpenVMS model.

Table 10–2: Unprivileged General PALcode Instruction Summary

Mnemonic	Operation
BPT	Breakpoint
BUGCHK	Bugcheck
CHME	Change mode to executive
CHMK	Change mode to kernel
CHMS	Change mode to supervisor
CHMU	Change mode to user
CLRFEN	Clear floating-point enable
GENTRAP	Generate software trap
IMB	I-stream memory barrier. See Section 6.7.3.
PROBER	Probe read access
PROBEW	Probe write access
RD_PS	Read processor status
REI	Return from exception or interrupt
RSCC	Read system cycle counter
SWASTEN	Swap AST enable
WR_PS_SW	Write processor status software field

10.1.1 Breakpoint

Format:

CALL_PAL BPT ! PALcode format

Operation:

{initiate BPT exception with new_mode=kernel}

Exceptions:

Kernel Stack Not Valid Halt

Instruction mnemonics:

CALL_PAL BPT Breakpoint

Description:

The BPT instruction is provided for program debugging. It switches 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 SCB vector. See Section 14.3.3.2.1.

10.1.2 Bugcheck

Format:

CALL_PAL BUGCHK ! PALcode format

Operation:

{initiate BUGCHK exception with new_mode=kernel}
! R16 contains a value encoding for the bugchk trap

Exceptions:

Kernel Stack Not Valid Halt

Instruction mnemonics:

CALL_PAL BUGCHK Bugcheck

Description:

The BUGCHK instruction is provided for error reporting. It switches 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 SCB vector. See Section 14.3.3.2.2.

The value in R16 identifies the particular bugcheck type. Interpretation of the encoded value determines the course of action by the operating system.

10.1.3 Change Mode to Executive

Format:

CALL_PAL CHME ! PALcode format

Operation:

```
tmp1 ← MINU( 1, PS<CM>)  
{initiate CHME exception with new_mode=tmp1}  
! R16 contains a value encoding for the trap
```

Exceptions:

Kernel Stack Not Valid Halt

Instruction mnemonics:

CALL_PAL CHME Change Mode to Executive

Description:

The CHME instruction lets a process 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. R2..R7, PC, and PS are pushed onto the selected stack. The saved PC addresses the instruction following the CHME instruction. Registers R22, R23, R24, and R27 are available for use by PALcode as scratch registers. The contents of these registers are not preserved across a CHME.

The value in R16 identifies the particular exception type. Interpretation of the encoded value determines the course of action by the operating system.

10.1.4 Change Mode to Kernel

Format:

CALL_PAL CHMK ! PALcode format

Operation:

{initiate CHMK exception with new_mode=kernel}
! R16 contains a value encoding for the trap

Exceptions:

Kernel Stack Not Valid Halt

Instruction mnemonics:

CALL_PAL CHMK Change Mode to Kernel

Description:

The CHMK instruction lets a process 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, PC, and PS are pushed onto the kernel stack. The saved PC addresses the instruction following the CHMK instruction. Registers R22, R23, R24, and R27 are available for use by PALcode as scratch registers. The contents of these registers are not preserved across a CHMK.

The value in R16 identifies the particular exception type. Interpretation of the encoded value determines the course of action by the operating system.

10.1.5 Change Mode to Supervisor

Format:

CALL_PAL CHMS ! PALcode format

Operation:

```
tmp1 ← MINU( 2, PS<CM>)  
{initiate CHMS exception with new_mode=tmp1}  
! R16 contains a value encoding for the trap
```

Exceptions:

Kernel Stack Not Valid Halt

Instruction mnemonics:

CALL_PAL CHMS Change Mode to Supervisor

Description:

The CHMS instruction lets a process 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. R2..R7, PC, and PS are pushed onto the selected stack. The saved PC addresses the instruction following the CHMS instruction.

The value in R16 identifies the particular exception type. Interpretation of the encoded value determines the course of action by the operating system.

10.1.6 Change Mode to User

Format:

CALL_PAL CHMU ! PALcode format

Operation:

{initiate CHMU exception with new_mode=PS<CM>}
! R16 contains a value encoding for the trap

Exceptions:

Kernel Stack Not Valid Halt

Instruction mnemonics:

CALL_PAL CHMU Change Mode to User

Description:

The CHMU instruction lets a process call a routine by using the change mode mechanism.

R2..R7, PC, and PS are pushed onto the current stack. The saved PC addresses the instruction following the CHMU instruction.

The value in R16 identifies the particular exception type. Interpretation of the encoded value determines the course of action by the operating system.

The CALL_PAL CHMU instruction is provided for VAX compatibility only.

10.1.7 Clear Floating-Point Enable

Format:

CALL_PAL CLRFEN ! PALcode format

Operation:

FEN \leftarrow 0
(HWPCB+56)<0> \leftarrow 0 ! Update HWPCB on Write

Exceptions:

None

Instruction mnemonics:

CALL_PAL CLRFEN Clear floating-point enable

Description:

The CLRFEN instruction writes a zero to the floating-point enable register and to the HWPCB at offset (HWPCB+56)<0>.

10.1.8 Generate Software Trap

Format:

CALL_PAL GENTRAP ! PALcode format

Operation:

{initiate GENTRAP exception with new_mode=kernel}
! R16 contains the value encoding of the software trap

Exceptions:

Kernel Stack Not Valid Halt

Instruction mnemonics:

CALL_PAL GENTRAP Generate Software Trap

Description:

The GENTRAP instruction is provided for reporting run-time software conditions. It switches to kernel mode and pushes R2...R7, the updated PC, and PS on the kernel stack. It then dispatches to the address in the GENTRAP SCB Vector. See Section 14.6.

The value in R16 identifies the particular software condition that has occurred. The encoding for the software trap values is given in the software calling standard for the system.

10.1.9 Probe Memory Access

Format:

CALL_PAL PROBE ! PALcode format

Operation:

```
! R16 contains the base address
! R17 contains the signed offset
! R18 contains the access mode
! R0 receives the completion status
!   ← 1 if success
!   ← 0 if failure
```

```
first ← R16
last  ← {R16+R17}
```

```
IF R18<1:0> GTU PS<CM> THEN
    probe_mode ← R18<1:0>
ELSE
    probe_mode ← PS<CM>
```

```
IF ACCESS(first, probe_mode) AND ACCESS(last, probe_mode) THEN
    R0 ← 1
ELSE
    R0 ← 0
```

Exceptions:

Translation Not Valid

Instruction mnemonics:

CALL_PAL	PROBER	Probe for Read Access
CALL_PAL	PROBEW	Probe for Write Access

Description:

The PROBE instruction checks the read or write 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. If both bytes are accessible, PROBE returns the value 1 in R0; otherwise, PROBE returns 0. The Fault on Read and Fault on Write PTE bits are not checked. A Translation Not Valid exception is signaled only if the mapping structures cannot be accessed. A Translation Not Valid exception is signaled only if a first- or second-level PTE is invalid.

The protection is checked against the less privileged of the modes specified by R18<1:0> and the Current Mode (PS<CM>). See Section 14.2 for access mode encodings.

PROBE is only intended to check a single datum for accessibility. It does not check all intervening pages because this could result in excessive interrupt latency.

10.1.10 Read Processor Status

Format:

CALL_PAL RD_PS ! PALcode format

Operation:

$R0 \leftarrow PS$

Exceptions:

None

Instruction mnemonics:

CALL_PAL RD_PS Read Processor Status

Description:

The RD_PS instruction returns the Processor Status (PS) in register R0. The Processor Status is described in Section 14.2. The PS<SP_ALIGN> field is always a zero on a RD_PS.

10.1.11 Return from Exception or Interrupt

Format:

CALL_PAL

REI

! PALcode format

Operation:

```
! See Chapter 14
! for information on interrupted registers

IF SP<5:0> NE 0 THEN
    {illegal operand }
tmp1 ← (SP)           ! Get saved R2
tmp2 ← (SP+8)        ! Get saved R3
tmp3 ← (SP+16)       ! Get saved R4
tmp4 ← (SP+24)       ! Get saved R5
tmp5 ← (SP+32)       ! Get saved R6
tmp6 ← (SP+40)       ! Get saved R7
tmp7 ← (SP+48)       ! Get new PC
tmp8 ← (SP+56)       ! Get new PS

ps_chk ← tmp8        ! Copy new ps
ps_chk<cm> ← 0        ! Clear cm field
ps_chk<sp_align> ← 0 ! Clear sp_align field
ps_chk<sw> ← 0        ! Clear Software Field
intr_flag ← 0        ! Clear except/inter/mcheck flag
{ clear lock_flag}

! If current mode is not kernel check the new ps is valid.
IF {ps<cm> NE 0} AND
    {{tmp8<cm> LT ps<cm>} OR {ps_chk NE 0}} THEN
    BEGIN
        {illegal operand}
    END

sp ← {sp + 8*8} OR tmp8<sp_align>
IF {internal registers for stack pointers} THEN
    CASE ps<cm> BEGIN
        [0]: ipr_ksp ← sp
        [1]: ipr_esp ← sp
        [2]: ipr_ssp ← sp
        [3]: ipr_usp ← sp
    ENDCASE
    CASE tmp8<cm> BEGIN
        [0]: sp ← ipr_ksp
        [1]: sp ← ipr_esp
        [2]: sp ← ipr_ssp
        [3]: sp ← ipr_usp
    ENDCASE
```


10.1.12 Read System Cycle Counter

Format:

CALL_PAL RSCC ! PALcode format

Operation:

R0 ← {System Cycle Counter}

Exceptions:

None

Instruction mnemonics:

CALL_PAL RSCC Read System Cycle Counter

Description:

The RSCC instruction writes register R0 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 cycle counter frequency, which is the number of times the system cycle counter gets incremented per second rounded to a 64-bit integer, is given in the HWRPB (see Section 26.1).

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. It is required on all implementations. SCC is required for every processor, and each processor in a multiprocessor system has its own private, independent SCC.

Notes:

- Processor initialization starts the SCC at 0.
- SCC is monotonically increasing. On the same processor, the values returned by two successive reads of SCC must either be equal or the value of the second must be greater (unsigned) than the first.
- SCC ticks are never lost so long as the SCC is accessed at least once per each PCC overflow period ($2^{*}32$ PCC increments) during periods when the hardware clock interrupt remains blocked. The hardware clock interrupt is blocked whenever the IPL is at or above CLOCK_IPL or whenever the processor enters console I/O mode from program I/O mode.
- The 64-bit SCC may be constructed from the 32-bit PCC hardware counter and a 32-bit PALcode software counter. As part of the hardware clock interrupt processing, PALcode increments the software counter whenever a PCC wrap is detected. Thus, SCC ticks may be lost only when PALcode fails to detect PCC wraps. In a machine where the PCC is incremented at a 1 ns rate, this may occur when hardware clock interrupts are blocked for greater than 4 seconds.

- An implementation-dependent mechanism must exist so that, when enabled, it causes the RSCC instruction, as implemented by standard PALcode, always to return a zero in R0. This mechanism must be usable by privileged system software. A similar mechanism must exist for RPCC. Implementations are allowed to have only a single mechanism, which when enabled causes both RSCC and RPCC to return zero.

10.1.13 Swap AST Enable

Format:

CALL_PAL SWASTEN ! PALcode format

Operation:

$R0 \leftarrow ZEXT(ASTEN<PS<CM>>)$
 $ASTEN<PS<CM>> \leftarrow R16<0>$

{check for pending ASTs}

Exceptions:

None

Instruction mnemonics:

CALL_PAL SWASTEN Swap AST Enable for Current Mode

Description:

The SWASTEN instruction swaps the AST enable bit for the current mode. The new state for the enable bit is supplied in register R16<0>, and the previous state of the enable bit is returned, zero extended, in R0.

A check is made to determine if an AST interrupt is pending (see Section 14.7.6.5).

If the enabling conditions are present for an AST interrupt at the completion of this instruction, the AST occurs before the next instruction.

10.1.14 Write Processor Status Software Field

Format:

CALL_PAL WR_PS_SW ! PALcode format

Operation:

PS<SW> ← R16<1:0>

Exceptions:

None

Instruction mnemonics:

CALL_PAL WR_PS_SW Write Processor Status Software Field

Description:

The WR_PS_SW instruction writes the Processor Status software field (PS<SW>) with the low-order two bits of R16. The Processor Status is described in Section 14.2.

10.2 Queue Data Types

The following sections describe the queue data types that are manipulated by the OpenVMS queue PALcode. Section 10.3 describes the PALcode instructions that perform the manipulation.

10.2.1 Absolute Longword Queues

A longword queue is a circular, doubly linked list. A longword queue entry is specified by its address. Each longword queue entry is linked to the next with a pair of longwords. A queue is classified by the type of link it uses. Absolute longword queues use absolute addresses as links.

The first (lowest addressed) longword is the forward link; it specifies the address of the succeeding longword queue entry. The second (highest addressed) longword is the backward link; it specifies the address of the preceding longword queue entry.

A longword queue is specified by a longword queue header, which is identical to a pair of longword queue linkage longwords. The forward link of the header is the address of the entry termed the head of the longword queue. The backward link of the header is the address of the entry termed the tail of the longword queue. The forward link of the tail points to the header.

An empty longword queue is specified by its header at address H, as shown in Figure 10–1. If an entry at address B is inserted into an empty longword queue (at either the head or tail), the longword queue shown in Figure 10–2 results. Figures 10–3, 10–4, and 10–5, respectively, illustrate the results of subsequent insertion of an entry at address A at the head, insertion of an entry at address C at the tail, and removal of the entry at address B.

The queue header and all entries in absolute longword queues need only be byte aligned. For better performance, quadword alignment (or higher) is recommended.

10.2.2 Self-Relative Longword Queues

Self-relative longword queues use displacements from longword queue entries as links. Longword queue entries are linked by a pair of longwords. The first longword (lowest addressed) is the forward link; it is a displacement of the succeeding longword queue entry from the present entry. The second longword (highest addressed) is the backward link; it is the displacement of the preceding longword queue entry from the present entry. A longword queue is specified by a longword queue header, which also consists of two longword links.

An empty longword queue is specified by its header at address H. Since the longword queue is empty, the self-relative links are zero, as shown in Figure 10–6.

Four types of operations can be performed on self-relative queues: insert at head, insert at tail, remove from head, and remove from tail. Furthermore, these operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared list without additional synchronization. A hardware-supported, interlocked memory-access mechanism is used to modify the queue header. Bit <0> of the queue header is used as a secondary interlock and is set when the queue is being accessed.

If an interlocked queue CALL_PAL instruction encounters the secondary interlock set, then, in the absence of exceptions, it terminates after setting R0 to -1 to indicate failure to gain access to the queue. If the secondary interlock bit is not set, then it is set during the interlocked queue operation and is cleared upon completion of the operation. This prevents other interlocked queue CALL_PAL instructions from operating on the same queue.

If both the secondary interlock is set and an exception condition occurs, it is UNPREDICTABLE whether the exception will be reported.

The queue header and all entries in self-relative longword queues must be at least quadword aligned.

Figures 10-7, 10-8, and 10-9, respectively, illustrate the results of subsequent insertion of an entry at address B at the head, insertion of an entry at address A at the tail, and insertion of an entry at address C at the tail.

Figures 10-9, 10-8, and 10-7 (in that order) illustrate the effect of removal at the tail and removal at the head.

Figure 10-1: Empty Absolute Longword Queue

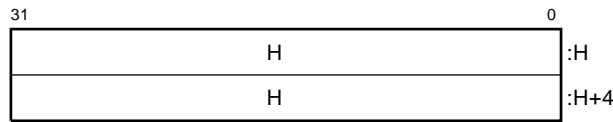


Figure 10-2: Absolute Longword Queue with One Entry

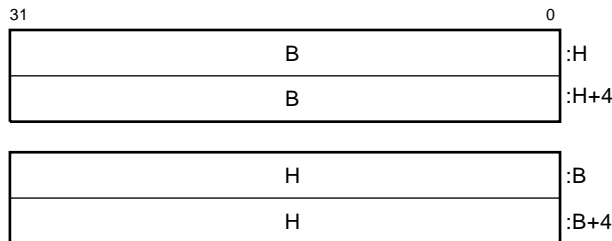


Figure 10-3: Absolute Longword Queue with Two Entries

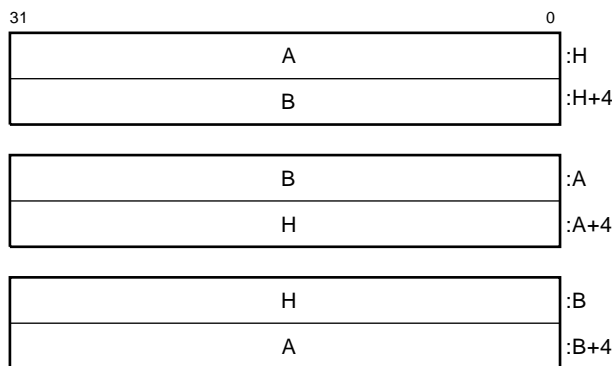


Figure 10–4: Absolute Longword Queue with Three Entries

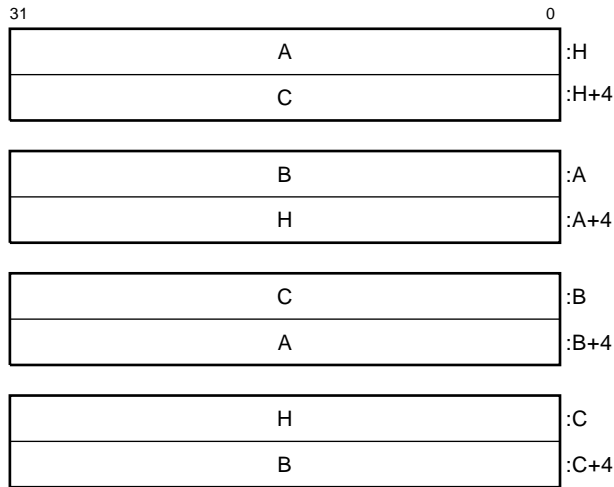


Figure 10–5: Absolute Longword Queue with Three Entries After Removing the Second Entry

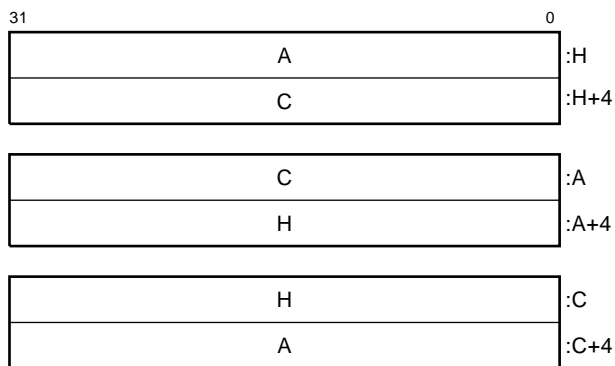


Figure 10–6: Empty Self-Relative Longword Queue

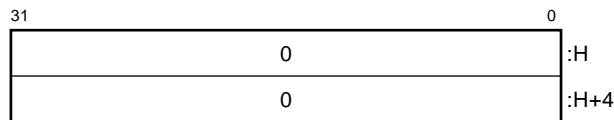


Figure 10–7: Self-Relative Longword Queue with One Entry

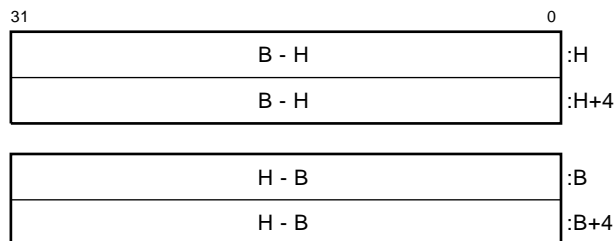


Figure 10–8: Self-Relative Longword Queue with Two Entries

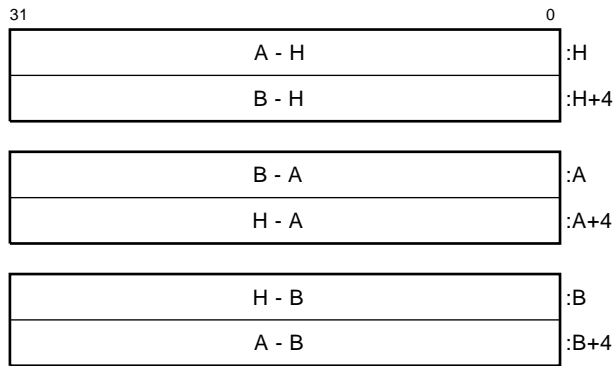
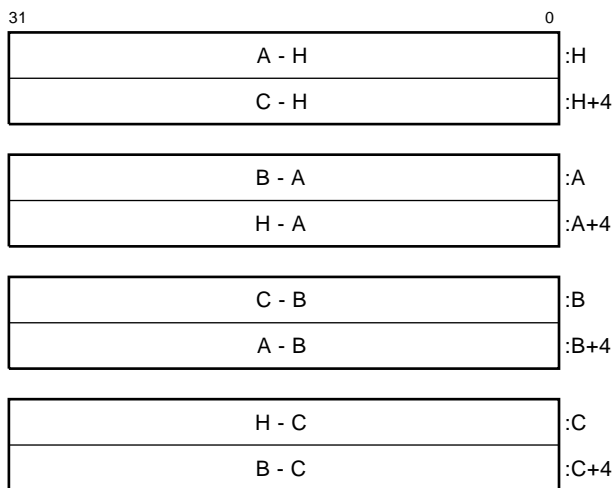


Figure 10–9: Self-Relative Longword Queue with Three Entries



10.2.3 Absolute Quadword Queues

A quadword queue is a circular, doubly linked list. A quadword queue entry is specified by its address. Each quadword queue entry is linked to the next with a pair of quadwords. A queue is classified by the type of link it uses. Absolute quadword queues use absolute addresses as links.

The first (lowest addressed) quadword is the forward link; it specifies the address of the succeeding quadword queue entry. The second (highest addressed) quadword is the backward link; it specifies the address of the preceding quadword queue entry.

A quadword queue is specified by a quadword queue header, which is identical to a pair of quadword queue linkage quadwords. The forward link of the header is the address of the entry termed the head of the quadword queue. The backward link of the header is the address of the entry termed the tail of the quadword queue. The forward link of the tail points to the header.

An empty quadword queue is specified by its header at address H, as shown in Figure 10–10. If an entry at address B is inserted into an empty quadword queue (at either the head or tail), the quadword queue shown in Figure 10–11 results. Figures 10–12, 10–13, and 10–14, respectively, illustrate the results of subsequent insertion of an entry at address A at the head, insertion of an entry at address C at the tail, and removal of the entry at address B.

The queue header and all entries in absolute quadword queues must be at least octaword aligned.

10.2.4 Self-Relative Quadword Queues

Self-relative quadword queues use displacements from quadword queue entries as links. Quadword queue entries are linked by a pair of quadwords. The first quadword (lowest addressed) is the forward link; it is a displacement of the succeeding quadword queue entry from the present entry. The second quadword (highest addressed) is the backward link; it is the displacement of the preceding quadword queue entry from the present entry. A quadword queue is specified by a quadword queue header, which also consists of two quadword links.

An empty quadword queue is specified by its header at address H. Since the quadword queue is empty, the self-relative links are zero, as shown in Figure 10–15.

Four types of operations can be performed on self-relative queues: insert at head, insert at tail, remove from head, and remove from tail. Furthermore, these operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared list without additional synchronization. A hardware-supported, interlocked memory-access mechanism is used to modify the queue header. Bit <0> of the queue header is used as a secondary interlock and is set when the queue is being accessed.

If an interlocked queue CALL_PAL instruction encounters the secondary interlock set, then, in the absence of exceptions, it terminates after setting R0 to –1 to indicate failure to gain access to the queue. If the secondary interlock bit is not set, it is set during the interlocked queue operation and is cleared upon completion of the operation. This prevents other interlocked queue CALL_PAL instructions from operating on the same queue.

If both the secondary interlock is set and an exception condition occurs, it is UNPREDICTABLE whether the exception will be reported.

The queue header and all entries in self-relative quadword queues must be at least octaword aligned.

Figures 10–16, 10–17, and 10–18, respectively, illustrate the results of subsequent insertion of an entry at address B at the head, insertion of an entry at address A at the tail, and insertion of an entry at address C at the tail.

Figures 10–18, 10–17, and 10–16 (in that order) illustrate the effect of removal at the tail and removal at the head.

Figure 10–10 Empty Absolute Quadword Queue

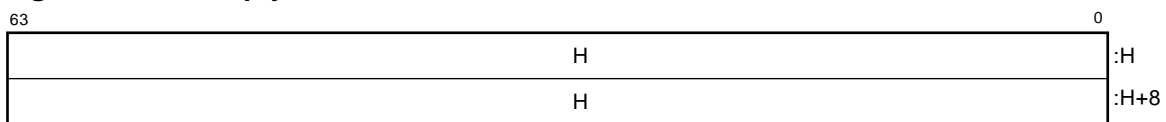


Figure 10–11 Absolute Quadword Queue with One Entry

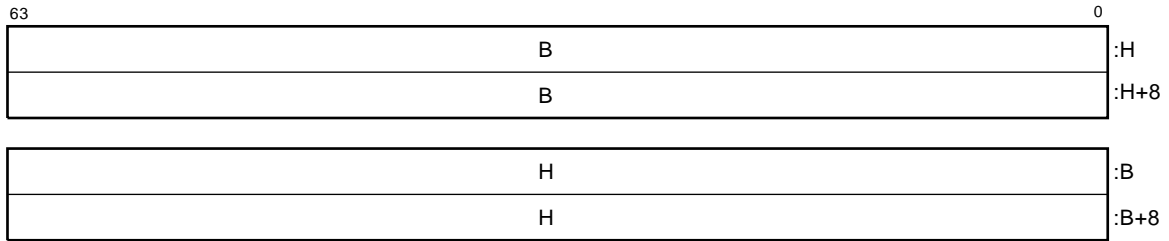


Figure 10–12 Absolute Quadword Queue with Two Entries

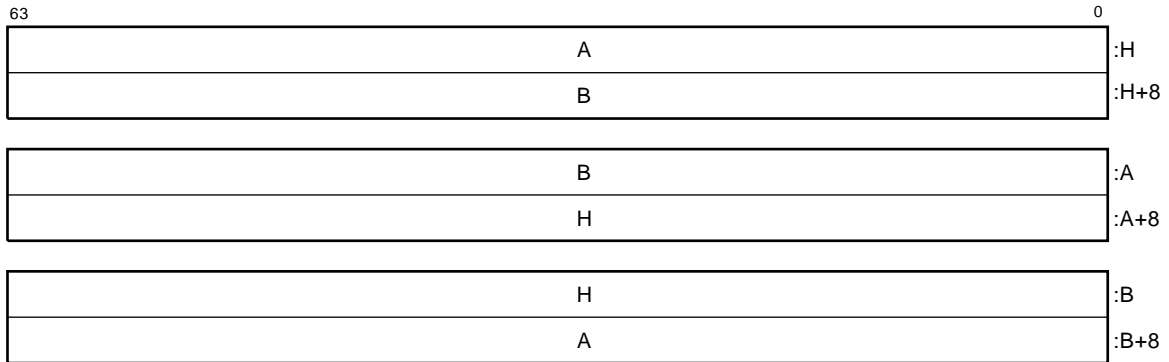


Figure 10–13 Absolute Quadword Queue with Three Entries

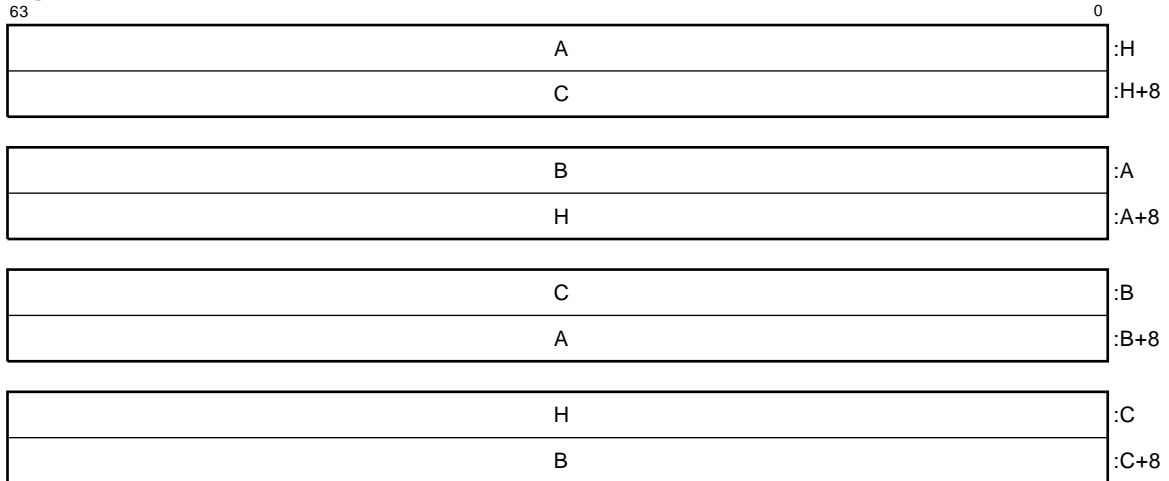


Figure 10–14 Absolute Quadword Queue with Three Entries After Removing the Second Entry

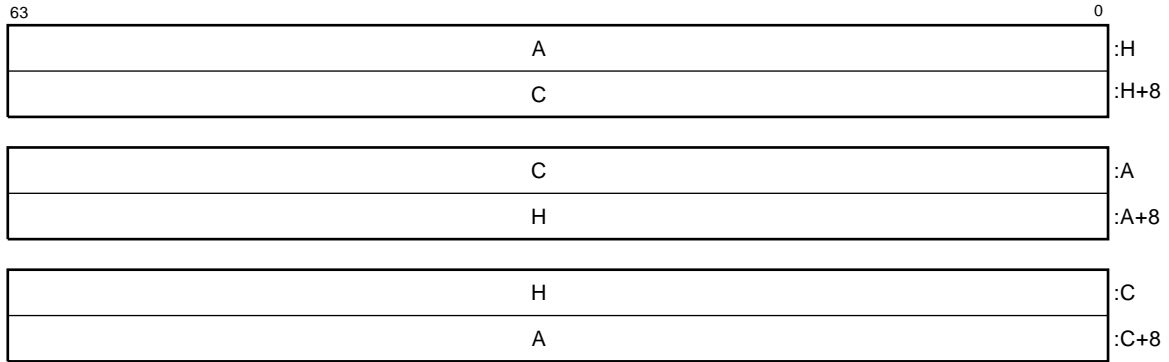


Figure 10–15 Empty Self-Relative Quadword Queue

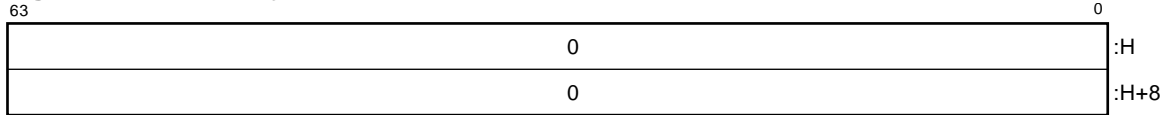


Figure 10–16 Absolute Quadword Queue with One Entry

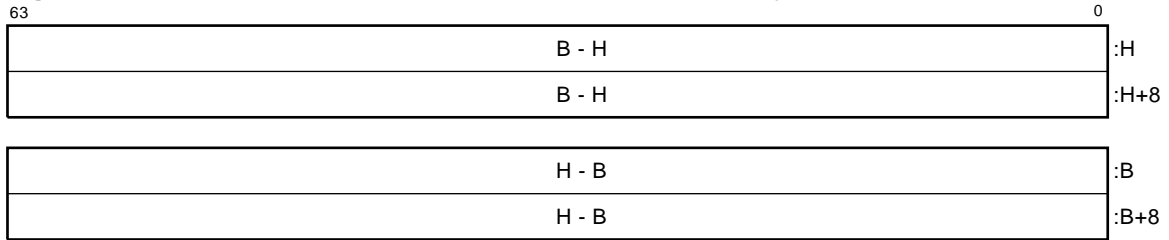


Figure 10–17 Self-Relative Quadword Queue with Two Entries

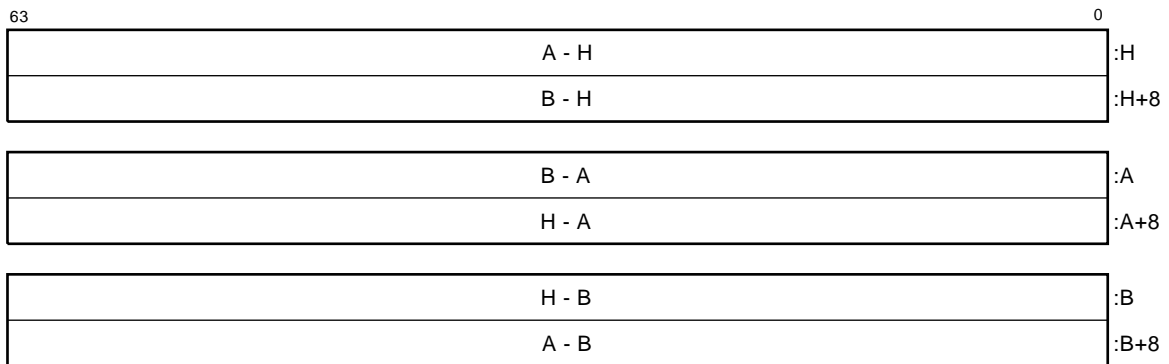
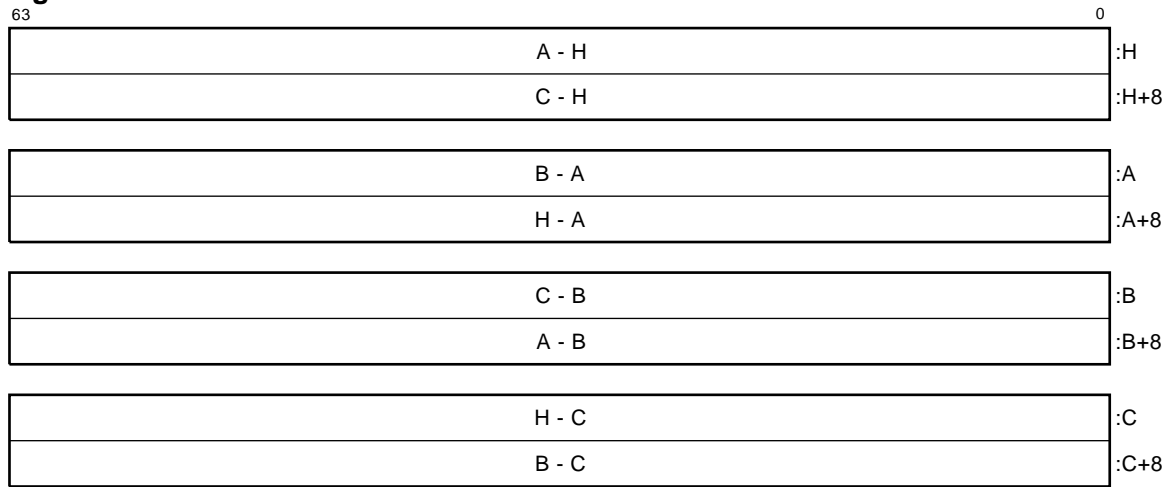


Figure 10–18 Self-Relative Quadword Queue with Three Entries



10.3 Unprivileged Queue PALcode Instructions

The following unprivileged PALcode instructions perform atomic modification of the queue data types that are described in Section 10.2.

Table 10–3: Queue PALcode Instruction Summary

Mnemonic	Operation
INSQHIL	Insert into longword queue at head, interlocked
INSQHILR	Insert into longword queue at head, interlocked, resident
INSQHIQ	Insert into quadword queue at head, interlocked
INSQHIQR	Insert into quadword queue at head, interlocked, resident
INSQTIL	Insert into longword queue at tail, interlocked
INSQTILR	Insert into longword queue at tail, interlocked, resident
INSQTIQ	Insert into quadword queue at tail, interlocked
INSQTIQR	Insert into quadword queue at tail, interlocked, resident
INSQUEL	Insert into longword queue
INSQUEQ	Insert into quadword queue
REMQHIL	Remove from longword queue at head, interlocked
REMQHILR	Remove from longword queue at head, interlocked, resident
REMQHIQ	Remove from quadword queue at head, interlocked
REMQHIQR	Remove from quadword queue at head, interlocked, resident
REMQTIL	Remove from longword queue at tail, interlocked
REMQTILR	Remove from longword queue at tail, interlocked, resident
REMQTIQ	Remove from quadword queue at tail, interlocked
REMQTIQR	Remove from quadword queue at tail, interlocked, resident
REMQUEL	Remove from longword queue
REMQUEQ	Remove from quadword queue

10.3.1 Insert Entry into Longword Queue at Head Interlocked

Format:

CALL_PAL INSQHIL ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!   -1 if the secondary interlock was set
!   0 if the queue was not empty before adding this entry
!   1 if the queue was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
! Header cannot be equal to entry.
!
! check entry and header alignment and
! that the header and entry not same location and
! that the header and entry are valid 32 bit addresses

IF {R16<2:0> NE 0} OR {R17<2:0> NE 0} OR {R16 EQ R17} OR
   {SEXT(R16<31:0>) NE R16} OR {SEXT(R17<31:0>) NE R17} THEN
  BEGIN
    {illegal operand exception}
  END

N <- {retry_amount}                      ! Implementation-specific
REPEAT
  LOAD_LOCKED (tmp0 ← (R16))! Acquire hardware interlock.
  IF tmp0<0> EQ 1 THEN                   ! Try to set secondary interlock
    R0 ← -1, {return}                   ! Already set
    done ←STORE_CONDITIONAL ((R16) ←{tmp0 OR 1} )
    N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB
tmp1 ← SEXT(tmp0<31:0>)
IF {tmp1<2:1> NE 0} THEN BEGIN         ! Check alignment
  BEGIN                                 ! Release secondary interlock.
    (R16) ← tmp0
    {illegal operand exception}
  END

! Check if following addresses can be written
! without causing a memory management exception:
!                   entry
!                   header + tmp1
```

```

IF {all memory accesses can NOT be completed} THEN
  BEGIN
    (R16) ← tmp0
    {initiate memory management fault}
  END

! All accesses can be done so enqueue the entry

tmp2 ← SEXT({R16 - R17}<31:0>)
(R17)<31:0> ← tmp1 + tmp2 ! Forward link
(R17 + 4)<31:0> ← tmp2 ! Backward link
(R16 + tmp1 + 4)<31:0> ← -tmp1 - tmp2! Successor back link

MB

(R16)<31:0> ← -tmp2 ! Forward link of header
! Release lock

IF tmp1 EQ 0 THEN
  R0 ← 1 ! Queue was empty
ELSE
  R0 ← 0 ! Queue was not empty
END

```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Illegal Operand
- Translation Not Valid

Instruction mnemonics:

CALL_PAL INSQHIL Insert into Longword Queue at Head Interlocked

Description:

If the secondary interlock is clear, INSQHIL inserts the entry specified in R17 into the self-relative queue following the header specified in R16.

If the entry inserted was the first one in the queue, R0 is set to 1; otherwise it is set to 0. The insertion is a non-interruptible 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. Before the insertion, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14). If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

10.3.2 Insert Entry into Longword Queue at Head Interlocked Resident

Format:

CALL_PAL INSQHILR ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!   -1 if the secondary interlock was set
!   0 if the queue was not empty before adding this entry
!   1 if the queue was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
! Header cannot be equal to entry.
! All parts of the Queue must be memory resident

N <- {retry_amount}            ! Implementation-specific
REPEAT
  LOAD_LOCKED (tmp0 <- (R16))! Acquire hardware interlock.
  IF tmp0<0> EQ 1 THEN           ! Try to set secondary interlock.
    R0 <- -1, {return}           ! Already set
    done <-STORE_CONDITIONAL ((R16) <-{tmp0 OR 1} )
    N <- N - 1
  UNTIL {done EQ 1} OR {N EQ 0}
  IF done NEQ 1, R0 <- -1, {return} ! Retry exceeded

MB

tmp1 <-SEXT(tmp0<31:0>)
tmp2 <-SEXT({R16 - R17}<31:0>)! Enqueue the entry
(R17)<31:0> <-tmp1 + tmp2    ! Forward link of entry.
(R17 + 4)<31:0> <-tmp2      ! Backward link of entry.
(R16 + tmp1 + 4)<31:0> <- -tmp1 - tmp2 ! Successor back link

MB
(R16)<31:0> <- -tmp2        ! Forward link of header
                            ! Release the lock

IF tmp1 EQ 0 THEN
  R0 <- 1                    ! Queue was empty
ELSE
  R0 <- 0                    ! Queue was not empty
END
```

Exceptions:

Illegal Operand

Instruction mnemonics:

CALL_PAL	INSQHILR	Insert Entry into Longword Queue at Head Interlocked Resident
----------	----------	--

Description:

If the secondary interlock is clear, INSQHILR inserts the entry specified in R17 into the self-relative queue following the header specified in R16.

If the entry inserted was the first one in the queue, R0 is set to 1; otherwise, it is set to 0. The insertion is a non-interruptible 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. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

This instruction requires that the queue be memory resident and that the queue header and elements are quadword aligned. No alignment or memory management checks are made before starting queue modifications to verify these requirements. Therefore, if any of these requirements are not met, the queue may be left in an UNPREDICTABLE state and an illegal operand fault may be reported.

10.3.3 Insert Entry into Quadword Queue at Head Interlocked

Format:

CALL_PAL INSQHIQ ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the entry was not empty before adding this entry
!     1 if the entry was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be octaword aligned.
! Header cannot be equal to entry.
!
! check entry and header alignment and
! that the header and entry not same location
IF {R16<3:0> NE 0} OR {R17<3:0> NE 0} OR {R16 EQ R17} THEN
    BEGIN
        {illegal operand exception}
    END

N <- {retry_amount}                      ! Implementation-specific
REPEAT
    LOAD_LOCKED (tmp1 ← (R16)) ! Acquire hardware interlock.
    IF tmp1<0> EQ 1 THEN                      ! Try to set secondary interlock.
        R0 ← -1, {return}                      ! Already set
        done ← STORE_CONDITIONAL ((R16) ←{tmp1 OR 1} )
        N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

IF {tmp1<3:1> NE 0} THEN BEGIN ! Check Alignment
    BEGIN                      ! Release secondary interlock
        (R16) ← tmp1
        {illegal operand exception}
    END

! Check if following addresses can be written
! without causing a memory management exception:
!     entry
!     header + tmp1
```

```

IF {all memory accesses can NOT be completed} THEN
  BEGIN
    (R16) ← tmp1
    {initiate memory management fault}
  END

! All accesses can be done so enqueue the entry
tmp2 ← R16 - R17
(R17) ← tmp1 + tmp2          ! Forward link
(R17 + 8) ← tmp1             ! Backward link
(R16 + tmp1 + 8) ← -tmp1 - tmp2 ! Successor back link

MB

(R16) ← -tmp2                ! Forward link of header
                              ! Release the lock.

IF tmp1 EQ 0 THEN
  R0 ← 1                      ! Queue was empty
ELSE
  R0 ← 0                      ! Queue was not empty
END

```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Illegal Operand
- Translation Not Valid

Instruction mnemonics:

CALL_PAL	INSQHIQ	Insert into Quadword Queue at Head
		Interlocked

Description:

If the secondary interlock is clear, INSQHIQ inserts the entry specified in R17 into the self-relative queue following the header specified in R16.

If the entry inserted was the first one in the queue, R0 is set to 1; otherwise, it is set to 0. The insertion is a non-interruptible 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. Before the insertion, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14). If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

10.3.4 Insert Entry into Quadword Queue at Head Interlocked Resident

Format:

CALL_PAL INSQHIQR ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the entry was not empty before adding this entry
!     1 if the entry was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be octaword aligned.
! Header cannot be equal to entry.
! All parts of the Queue must be memory resident

N <- {retry_amount}                      ! Implementation-specific
REPEAT
    LOAD_LOCKED (tmp1 ← (R16))            ! Acquire hardware interlock.
    IF tmp1<0> EQ 1 THEN                    ! Try to set secondary interlock.
        R0 ← -1, {return}                    ! Already set
        done ←STORE_CONDITIONAL ((R16) ←{tmp1 OR 1} )
        N ← N - 1
    UNTIL {done EQ 1} OR {N EQ 0}
    IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

tmp2 ← R16 - R17                            ! Enqueue the entry
(R17) ← tmp1 + tmp2                        ! Forward link of entry.
(R17 + 8) ← tmp2                            ! Backward link of entry.
(R16 + tmp1 + 8) ← -tmp1 - tmp2 ! Successor back link

MB
(R16) ← -tmp2                                ! Forward link of header,
                                              ! Release the lock

IF tmp1 EQ 0 THEN
    R0 ← 1                                    ! Queue was empty
ELSE
    R0 ← 0                                    ! Queue was not empty
END
```

Exceptions:

Illegal Operand

Instruction mnemonics:

CALL_PAL INSQHIQR Insert Entry into Quadword Queue at Head
Interlocked Resident

Description:

If the secondary interlock is clear, INSQHIQR inserts the entry specified in R17 into the self-relative queue following the header specified in R16.

If the entry inserted was the first one in the queue, R0 is set to 1; otherwise, it is set to 0. The insertion is a non-interruptible 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. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

This instruction requires that the queue be memory resident and that the queue header and elements are octaword aligned. No alignment or memory management checks are made before starting queue modifications to verify these requirements. Therefore, if any of these requirements are not met, the queue may be left in an UNPREDICTABLE state and an illegal operand fault may be reported.

10.3.5 Insert Entry into Longword Queue at Tail Interlocked

Format:

```
CALL_PAL          INSQTL          ! PALcode format
```

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!   -1 if the secondary interlock was set
!   0 if the entry was not empty before adding this entry
!   1 if the entry was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
! Header cannot be equal to entry.
!
! check entry and header alignment and
! that the header and entry not same location and
! that the header and entry are valid 32 bit addresses
IF {R16<2:0> NE 0} OR {R17<2:0> NE 0} OR {R16 EQ R17} OR
   {SEXT(R16<31:0>) NE R16} OR {SEXT(R17<31:0>) NE R16} THEN
  BEGIN
    {illegal operand exception}
  END
N <- {retry_amount}          ! Implementation-specific
REPEAT
  LOAD_LOCKED (tmp0 <- (R16)) ! Acquire hardware interlock.
  IF tmp0<0> EQ 1 THEN        ! Try to set secondary interlock.
    R0 <- -1, {return}       ! Already set
    done <- STORE_CONDITIONAL ((R16) <-{tmp0 OR 1} )
    N <- N - 1
  UNTIL {done EQ 1} OR {N EQ 0}
  IF done NEQ 1, R0 <- -1, {return} ! Retry exceeded

MB

tmp1 <- SEXT(tmp0<31:0>)
tmp2 <- SEXT(tmp0<63:32>)

IF {tmp1<2:1> NE 0} OR {tmp2<2:0> NE 0} THEN ! Check Alignment
  BEGIN                                     ! Release secondary interlock
    (R16) <- tmp0
    {illegal operand exception}
  END

! Check if following addresses can be written
! without causing a memory management exception:
!   entry
!   header + (header + 4)
```

```

IF {all memory accesses can NOT be completed} THEN
  BEGIN
    (R16) ← tmp0
    {initiate memory management fault}
  END

! All Accesses can be done so enqueue entry
tmp3 ← SEXT( {R16 - R17}<31:0>)
(R17)<31:0> ← tmp3 ! Forward link
(R17 + 4)<31:0> ← tmp2 + tmp3 ! Backward link
IF {tmp2 NE 0} THEN ! Forward link of predecessor
  (R16+tmp2)<31:0> ← -tmp3 - tmp2
ELSE
  tmp1 ← SEXT({-tmp3 - tmp2}<31:0>)
  (R16+4)<31:0> ← -tmp3 ! Backward link of header
MB
(R16)<31:0> ← tmp1 ! Forward link, release lock
IF tmp1 EQ -tmp3 THEN
  R0 ← 1 ! Queue was empty
ELSE
  R0 ← 0 ! Queue was not empty
END

```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Illegal Operand
- Translation Not Valid

Instruction mnemonics:

CALL_PAL INSQTIL Insert into Longword Queue at Tail Interlocked

Description:

If the secondary interlock is clear, INSQTIL inserts the entry specified in R17 into the self-relative queue preceding the header specified in R16.

If the entry inserted was the first one in the queue, R0 is set to 1; otherwise, it is set to 0. The insertion is a non-interruptible 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. Before performing any part of the operation, the processor validates that the insertion can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14). If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

10.3.6 Insert Entry into Longword Queue at Tail Interlocked Resident

Format:

CALL_PAL INSQTILR ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!   -1 if the secondary interlock was set
!   0 if the entry was not empty before adding this entry
!   1 if the entry was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
! Header cannot be equal to entry.
! All parts of the Queue must be memory resident

N <- {retry_amount}                      ! Implementation-specific
REPEAT
  LOAD_LOCKED (tmp0 <- (R16))           ! Acquire hardware interlock.
  IF tmp0<0> EQ 1 THEN                   ! Try to set secondary interlock.
    R0 <- -1, {return}                   ! Already set
    done <- STORE_CONDITIONAL ((R16) <- {tmp0 OR 1} )
    N <- N - 1
  UNTIL {done EQ 1} OR {N EQ 0}
  IF done NEQ 1, R0 <- -1, {return} ! Retry exceeded

MB

tmp1 <- SEXT(tmp0<31:0>)
tmp2 <- SEXT(tmp0<63:32>)
tmp3 <- SEXT( {R16 - R17}<31:0>)
(R17)<31:0> <- tmp3                      ! Forward link
(R17 + 4)<31:0> <- tmp2 + tmp3           ! Backward link
IF {tmp2 NE 0} THEN                      ! Forward link of predecessor
  (R16+tmp2)<31:0> <- -tmp3 - tmp2
ELSE
  tmp1 <- <- SEXT({-tmp3 - tmp2}<31:0>)

(R16+4)<31:0> <- -tmp3                   ! Backward link of header

MB

(R16)<31:0> <- tmp1                      ! Forward link
                                        ! Release the lock
```

```

IF tmp1 EQ -tmp3 THEN
    R0 ← 1                ! Queue was empty
ELSE
    R0 ← 0                ! Queue was not empty
END

```

Exceptions:

Illegal Operand

Instruction mnemonics:

```

CALL_PAL  INSQTILR      Insert Entry into Longword Queue at Tail
                          Interlocked Resident

```

Description:

If the secondary interlock is clear, INSQTILR inserts the entry specified in R17 into the self-relative queue preceding the header specified in R16.

If the entry inserted was the first one in the queue, R0 is set to 1; otherwise, it is set to 0. The insertion is a non-interruptible 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. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

This instruction requires that the queue be memory resident and that the queue header and elements are quadword aligned. No alignment or memory management checks are made before starting queue modifications to verify these requirements. Therefore, if any of these requirements are not met, the queue may be left in an UNPREDICTABLE state and an illegal operand fault may be reported.

10.3.7 Insert Entry into Quadword Queue at Tail Interlocked

Format:

CALL_PAL INSQTIQ ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!   -1 if the secondary interlock was set
!   0 if the entry was not empty before adding this entry
!   1 if the entry was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be octaword aligned.
! Header cannot be equal to entry.
!
! check entry and header alignment and
! that the header and entry not same location
IF {R16<3:0> NE 0} OR {R17<3:0> NE 0} OR {R16 EQ R17} THEN
  BEGIN
    {illegal operand exception}
  END

N <- {retry_amount}                      ! Implementation-specific
REPEAT
  LOAD_LOCKED (tmp1 ← (R16))    ! Acquire hardware interlock.
  IF tmp1<0> EQ 1 THEN            ! Try to set secondary interlock.
    R0 ← -1, {return}            ! Already set
    done ← STORE_CONDITIONAL ((R16) ← {tmp1 OR 1} )
    N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

tmp2 ← (R16+8)
IF {tmp1<3:1> NE 0} OR {tmp2<3:0> NE 0} THEN ! Check Alignment.
  BEGIN                            ! Release secondary interlock.
    (R16) ← tmp1
    {illegal operand exception}
  END

! Check if following addresses can be written
! without causing a memory management exception:
!            entry
!            header + (header + 8)
```

```

IF {all memory accesses can NOT be completed} THEN
  BEGIN
    (R16) ← tmp1
    {initiate memory management fault}
  END
  ! All accesses can be done so enqueue the entry
tmp3 ← R16 - R17
(R17) ← tmp3
(R17 + 8) ← tmp2 + tmp3
IF {tmp2 NE 0} THEN
  (R16+tmp2) ← -tmp3 - tmp2
ELSE
  tmp1 ← {-tmp3 - tmp2}
(R16+8) ← -tmp3
MB

(R16) ← tmp1
IF tmp1 EQ -tmp3 THEN
  R0 ← 1
ELSE
  R0 ← 0
END

```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Illegal Operand
- Translation Not Valid

Instruction mnemonics:

CALL_PAL INSQTIQ Insert into Quadword Queue at Tail Interlocked

Description:

If the secondary interlock is clear, INSQTIQ inserts the entry specified in R17 into the self-relative queue preceding the header specified in R16.

If the entry inserted was the first one in the queue, R0 is set to 1; otherwise, it is set to 0. The insertion is a non-interruptible 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. Before performing any part of the operation, the processor validates that the insertion can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14). If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

10.3.8 Insert Entry into Quadword Queue at Tail Interlocked Resident

Format:

CALL_PAL INSQTIQR ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!   -1 if the secondary interlock was set
!   0 if the entry was not empty before adding this entry
!   1 if the entry was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be octaword aligned.
! Header cannot be equal to entry.
! All parts of the Queue must be memory resident

N <- {retry_amount}                      ! Implementation-specific
REPEAT
  LOAD_LOCKED (tmp1 ← (R16))            ! Acquire hardware interlock.
  IF tmp1<0> EQ 1 THEN                   ! Try to set secondary interlock.
    R0 ← -1, {return}                   ! Already set
    done ← STORE_CONDITIONAL ((R16) ← {tmp1 OR 1} )
    N ← N - 1
  UNTIL {done EQ 1} OR {N EQ 0}
  IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

tmp2 ← (R16+8)
tmp3 ← R16 - R17
(R17) ← tmp3                            ! Forward link
(R17 + 8) ← tmp2 + tmp3                ! Backward link
IF {tmp2 NE 0} THEN                    ! Forward link of predecessor
  (R16+tmp2) ← -tmp3 - tmp2
ELSE
  tmp1 ← {-tmp3 - tmp2}
  (R16+8) ← -tmp3                       ! Backward link of header

MB

(R16) ← tmp1                            ! Forward link and release the lock
IF tmp1 EQ -tmp3 THEN
  R0 ← 1                                ! Queue was empty
ELSE
  R0 ← 0                                ! Queue was not empty
END
```

Exceptions:

Illegal Operand

Instruction mnemonics:

CALL_PAL INSQTIQR Insert Entry into Quadword Queue at Tail
Interlocked Resident

Description:

If the secondary interlock is clear, INSQTIQR inserts the entry specified in R17 into the self-relative queue preceding the header specified in R16.

If the entry inserted was the first one in the queue, R0 is set to 1; otherwise, it is set to 0. The insertion is a non-interruptible 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. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

This instruction requires that the queue be memory resident and that the queue header and elements are octaword aligned. No alignment or memory management checks are made before starting queue modifications to verify these requirements. Therefore, if any of these requirements are not met, the queue may be left in an UNPREDICTABLE state and an illegal operand fault may be reported.

10.3.9 Insert Entry into Longword Queue

Format:

CALL_PAL INSQUEL ! PALcode format

Operation:

```
! R16 contains the address of the predecessor entry
!   or the 32 bit address of the 32 bit address of the
!   predecessor entry for INSQUEL/D
! R17 contains the address of the new entry
! R0 receives status:
!   0 if the queue was not empty before adding this entry
!   1 if the queue was empty before adding this entry
!
! Header and entries need only be byte aligned
! Must have write access to header and queue entries
IF opcode EQ INSQUEL/D THEN
    tmp2 ← SEXT((R16)<31:0>)! Address of predecessor
ELSE
    tmp2 ← R16

IF {all memory accesses can be completed} THEN
    BEGIN
        tmp1<31:0> ← SEXT((tmp2)<31:0>)! Get Forward Link
        (R17)<31:0> ← tmp1           ! Set forward link
        (R17 + 4)<31:0> ← tmp2 ! Backward link
        (SEXT((tmp2)<31:0>) + 4)<31:0> ← R17
                                   ! Backward link of Successor
        (tmp2)<31:0> ← R17           ! Forward link of Predecessor
    IF tmp1 EQ tmp2 THEN
        R0 ← 1
    ELSE
        R0 ← 0
    END
ELSE
    BEGIN
        {initiate fault}
    END
END
```

Exceptions:

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

Instruction mnemonics:

CALL_PAL	INSQUEL	Insert Entry into Longword Queue
CALL_PAL	INSQUEL/D	Insert Entry into Longword Queue Deferred

Description:

INSQUEL inserts the entry specified in R17 into the absolute queue following the entry specified by the predecessor addressed by R16. INSQUEL/D performs the same operation on the entry specified by the contents of the longword addressed by R16. The queue header and entry need only be byte aligned.

In either case, if the entry inserted was the first one in the queue, a 1 is returned in R0; otherwise, a 0 is returned in R0. The insertion is a non-interruptible operation. Before performing any part of the insertion, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14).

10.3.10 Insert Entry into Quadword Queue

Format:

CALL_PAL INSQUEQ ! PALcode format

Operation:

```
! R16 contains the address of the predecessor entry
!   or the address of the address of the
!   predecessor entry for INSQUEQ/D
! R17 contains the address of the new entry
! R0 receives status:
!   0 if the queue was not empty before adding this entry
!   1 if the queue was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be octaword aligned

IF opcode EQ INSQUEQ/D THEN
  IF {R16<3:0> NE 0} THEN
    BEGIN
      {illegal operand exception}
    END
    tmp2 ← (R16)                      ! Address of predecessor
  ELSE
    tmp2 ← R16
  END
  IF {tmp2<3:0> NE 0} OR {R17<3:0> NE 0} THEN
    BEGIN
      {illegal operand exception}
    END
    IF {all memory accesses can be completed} THEN
      BEGIN
        tmp1 ← (tmp2)                      ! Get forward link of entry
        IF {tmp1<3:0> NE 0} THEN
          BEGIN                      ! Check alignment
            {illegal operand exception}
          END
          (R17) ← tmp1                      ! Set forward link of entry
          (R17 + 8) ← tmp2                      ! Backward link of entry
          (tmp1 + 8) ← R17                      ! Backward link of successor
          (tmp2) ← R17                      ! Forward link of predecessor
        IF tmp1 EQ tmp2 THEN
          R0 ← 1
        ELSE
          R0 ← 0
        END
      END
    END
  END
END
```

```
ELSE
  BEGIN
    {initiate fault}
  END
END
```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Translation Not Valid
- Illegal Operand

Instruction mnemonics:

CALL_PAL	INSQUEQ	Insert Entry into Quadword Queue
CALL_PAL	INSQUEQ/D	Insert Entry into Quadword Queue Deferred

Description:

INSQUEQ inserts the entry specified in R17 into the absolute queue following the entry specified by the predecessor addressed by R16. INSQUEQ/D performs the same operation on the entry specified by the contents of the quadword addressed by R16.

In either case, if the entry inserted was the first one in the queue, a 1 is returned in R0; otherwise, a 0 is returned in R0. The insertion is a non-interruptible operation. Before performing any part of the insertion, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14). R0 is UNPREDICTABLE if an exception occurs. The relative order of reporting memory management and illegal operand exceptions is UNPREDICTABLE.

10.3.11 Remove Entry from Longword Queue at Head Interlocked

Format:

CALL_PAL REMQHIL ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the queue was empty
!     1 if entry removed and queue still not empty
!     2 if entry removed and queue empty
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
!
! Check header alignment and
! that the header is a valid 32 bit address
IF {R16<2:0> NE 0} OR {SEXT(R16<31:0>) NE R16} THEN
    BEGIN
        {illegal operand exception}
    END

N <- {retry_amount}                      ! Implementation-specific
REPEAT
    LOAD_LOCKED (tmp0 <- (R16))      ! Acquire hardware interlock.
    IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
        R0 <- -1, {return}            ! Already set
        done <- STORE_CONDITIONAL ((R16) <- {tmp0 OR 1} )
        N <- N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 <- -1, {return} ! Retry exceeded

MB

tmp1 <- SEXT(tmp0<31:0>)
IF tmp1<2:0> NE 0 THEN              ! Check Alignment
    BEGIN                              ! Release secondary interlock
        (R16) <- tmp0
        {illegal operand exception}
    END
```

```

! Check if the following can be done without
! causing a memory management exception:
! read contents of header + tmp1 {if tmp1 NE 0}
! write into header + tmp1 + (header + tmp1) {if tmp1 NE 0}
IF {all memory accesses can NOT be completed} THEN
  BEGIN
    ! Release secondary interlock
    (R16) ← tmp0
    {initiate memory management fault}
  END

tmp2 ← SEXT({R16 + tmp1}<31:0>)
IF {tmp1 EQL 0} THEN
  tmp3 ← R16
ELSE
  tmp3 ← SEXT({tmp2 + SEXT((tmp2)<31:0>)})

IF tmp3<2:0> NE 0 THEN
  ! Check Alignment
  BEGIN
    ! Release secondary interlock
    (R16) ← tmp0
    {illegal operand exception}
  END

(tmp3 + 4)<31:0> ← R16 - tmp3 ! Backward link of successor

MB

(R16)<31:0> ← tmp3 - R16
! Forward link of header
! Release lock

IF tmp1 EQ 0 THEN
  R0 ← 0
! Queue was empty
ELSE
  BEGIN
    IF {tmp3 - R16} EQ 0 THEN
      R0 ← 2
! Queue now empty
    ELSE
      R0 ← 1
! Queue not empty
    END
  END

R1 ← tmp2
! Address of removed entry

```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Illegal Operand
- Translation Not Valid

Instruction mnemonics:

CALL_PAL	REMQHIL	Remove from Longword Queue at Head
		Interlocked

Description:

If the secondary interlock is clear, REMQHIL removes from the self-relative queue the entry following the header, pointed to by R16, and the address of the removed entry is returned in R1.

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal and the queue is empty after the removal, a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

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 non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14).

10.3.12 Remove Entry from Longword Queue at Head Interlocked Resident

Format:

CALL_PAL REMQHILR ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the queue was empty
!     1 if entry removed and queue still not empty
!     2 if entry removed and queue empty
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
! All parts of the Queue must be memory resident

N <- {retry_amount}                      ! Implementation-specific
REPEAT
    LOAD_LOCKED (tmp0 <- (R16))        ! Acquire hardware interlock.
    IF tmp0<0> EQ 1 THEN                ! Try to set secondary interlock.
        R0 <- -1, {return}              ! Already set
        done <- STORE_CONDITIONAL ((R16) <- {tmp0 OR 1} )
        N <- N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 <- -1, {return} ! Retry exceeded

MB

tmp1 <- SEXT(tmp0<31:0>)
tmp2 <- SEXT({R16 + tmp1}<31:0>)
IF {tmp1 EQL 0} THEN
    tmp3 <- R16
ELSE
    tmp3 <- SEXT({tmp2 + SEXT((tmp2)<31:0>)})
END

(tmp3 + 4)<31:0> <- R16 - tmp3            ! Backward link of successor

MB
(R16)<31:0> <- tmp3 - R16                ! Forward link of header
                                        ! Release lock

IF tmp1 EQ 0 THEN
    R0 <- 0                              ! Queue was empty
```



```

ELSE
  BEGIN
    IF {tmp3 - R16} EQ 0 THEN
      R0 ← 2           ! Queue now empty
    ELSE
      R0 ← 1           ! Queue not empty
    END
  END
END
R1 ← tmp2           ! Address of removed entry

```

Exceptions:

Illegal Operand

Instruction mnemonics:

CALL_PAL	REMQHILR	Remove Entry from Longword Queue at Head Interlocked Resident
----------	----------	--

Description:

If the secondary interlock is clear, REMQHILR removes from the self-relative queue the entry following the header, pointed to by R16, and the address of the removed entry is returned in R1.

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal and the queue is empty after the removal, a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

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 non-interruptible operation.

This instruction requires that the queue be memory resident and that the queue header and elements are quadword aligned. No alignment or memory management checks are made before starting queue modifications to verify these requirements. Therefore, if any of these requirements are not met, the queue may be left in an UNPREDICTABLE state and an illegal operand fault may be reported.

10.3.13 Remove Entry from Quadword Queue at Head Interlocked

Format:

CALL_PAL REMQHIQ ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the queue was empty
!     1 if entry removed and queue still not empty
!     2 if entry removed and queue empty
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
! Header and entries must be octaword aligned.
!
! Check header alignment
IF {R16<3:0> NE 0} THEN
    BEGIN
        {illegal operand exception}
    END

N <- {retry_amount}                      ! Implementation-specific
REPEAT
    LOAD_LOCKED (tmp1 ← (R16))    ! Acquire hardware interlock.
    IF tmp1<0> EQ 1 THEN            ! Try to set secondary interlock.
        R0 ← -1, {return}            ! Already set
        done ← STORE_CONDITIONAL ((R16) ← {tmp1 OR 1} )
        N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

IF tmp1<3:0> NE 0 THEN            ! Check Alignment
    BEGIN                            ! Release secondary interlock
        (R16) ← tmp1
        {illegal operand exception}
    END

! Check if the following can be done without
! causing a memory management exception:
! read contents of header + tmp1 {if tmp1 NE 0}
! write into header + tmp1 + (header + tmp1) {if tmp1 NE 0}
```

```

IF {all memory accesses can NOT be completed} THEN
  BEGIN
    (R16) ← tmp0
    {initiate memory management fault}
  END

tmp2 ← R16 + tmp1
IF {tmp1 EQL 0} THEN
  tmp3 ← R16
ELSE
  tmp3 ← tmp2 + (tmp2)

IF tmp3<3:0> NE 0 THEN
  BEGIN
    (R16) ← tmp1
    {illegal operand exception}
  END

(tmp3 + 8) ← R16 - tmp3
! Backward link of successor

MB

(R16) ← tmp3 - R16
! Forward link of header
! Release lock

IF tmp1 EQ 0 THEN
  R0 ← 0
! Queue was empty
ELSE
  BEGIN
    IF {tmp3 - R16} EQ 0 THEN
      R0 ← 2
! Queue now empty
    ELSE
      R0 ← 1
! Queue not empty
    END
  END
END
R1 ← tmp2
! Address of removed entry

```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Illegal Operand
- Translation Not Valid

Instruction mnemonics:

CALL_PAL	REMQHIQ	Remove from Quadword Queue at Head
		Interlocked

Description:

If the secondary interlock is clear, REMQHIQ removes from the self-relative queue the entry following the header, pointed to by R16, and the address of the removed entry is returned in R1.

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal, a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

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 non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14).

10.3.14 Remove Entry from Quadword Queue at Head Interlocked Resident

Format:

CALL_PAL REMQHIQR ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the queue was empty
!     1 if entry removed and queue still not empty
!     2 if entry removed and queue empty
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
! Header and entries must be octaword aligned.
! All parts of the Queue must be memory resident

N ← {retry_amount}                      ! Implementation-specific
REPEAT
    LOAD_LOCKED (tmp1 ← (R16))        ! Acquire hardware interlock.
    IF tmp1<0> EQ 1 THEN                ! Try to set secondary interlock.
        R0 ← -1, {return}              ! Already set
        done ← STORE_CONDITIONAL ((R16) ← {tmp1 OR 1} )
        N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

tmp2 ← R16 + tmp1
IF {tmp1 EQL 0} THEN
    tmp3 ← R16
ELSE
    tmp3 ← tmp2 + (tmp2)
END
(tmp3 + 8) ← R16 - tmp3                ! Backward link of successor

MB

(R16) ← tmp3 - R16                      ! Forward link of header
                                        ! Release lock
```

```

IF tmp1 EQ 0 THEN
    R0 ← 0                                ! Queue was empty
ELSE
    IF {tmp3 - R16} EQ 0 THEN
        R0 ← 2                            ! Queue now empty
    ELSE
        R0 ← 1                            ! Queue not empty
    END
R1 ← tmp2                                ! Address of removed entry

```

Exceptions:

Illegal Operand

Instruction mnemonics:

CALL_PAL REMQHIQR Remove Entry from Quadword Queue at Head
Interlocked Resident

Description:

If the secondary interlock is clear, REMQHIQR removes from the self-relative queue the entry following the header, pointed to by R16, and the address of the removed entry is returned in R1.

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal, a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

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 non-interruptible operation.

This instruction requires that the queue be memory resident and that the queue header and elements are octaword aligned. No alignment or memory management checks are made before starting queue modifications to verify these requirements. Therefore, if any of these requirements are not met, the queue may be left in an UNPREDICTABLE state and an illegal operand fault may be reported.

10.3.15 Remove Entry from Longword Queue at Tail Interlocked

Format:

CALL_PAL REMQTIL ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the queue was empty
!     1 if entry removed and queue still not empty
!     2 if entry removed and queue empty
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
!
! Check header alignment and
! that the header is a valid 32 bit address
IF {R16<2:0> NE 0} OR {SEXT(R16<31:0>) NE R16} THEN
    BEGIN
        {illegal operand exception}
    END

N <- {retry_amount}                      ! Implementation-specific
REPEAT
    LOAD_LOCKED (tmp0 <- (R16))        ! Acquire hardware interlock.
    IF tmp0<0> EQ 1 THEN                ! Try to set secondary interlock.
        R0 <- -1, {return}             ! Already set
        done <- STORE_CONDITIONAL ((R16) <- {tmp0 OR 1} )
        N <- N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 <- -1, {return} ! Retry exceeded

MB

tmp1 <- SEXT(tmp0<31:0>)
tmp5 <- SEXT(tmp0<63:32>)
IF tmp5<2:0> NE 0 THEN                ! Check alignment
    BEGIN                              ! Release secondary interlock
        (R16) <- tmp0
        {illegal operand exception}
    END

!Check if the following can be done without
! causing a memory management exception:
! read contents of header + (header + 4) {if tmp1 NE 0}
! write into header + (header + 4)
! + (header + 4 + (header + 4)){if tmp1 NE 0}
```

```

IF {all memory accesses can NOT be completed} THEN
  BEGIN
    (R16) ← tmp0
    {initiate memory management fault}
  END

addr ← SEXT( {R16 + tmp5}<31:0> )
tmp2 ← SEXT( {addr + SEXT( (addr+4)<31:0>)}<31:0> )
IF tmp2<2:0> NE 0 THEN
  BEGIN
    (R16) ← tmp0
    {illegal operand exception}
  END

(R16 + 4)<31:0> ← tmp2 - R16 ! Backward link of header
IF {tmp2 EQL R16} THEN
  (R16)<31:0> ← 0 ! Forward link, release lock
ELSE
  BEGIN
    (tmp2)<31:0> ← R16 - tmp2 ! Forward link of predecessor
  MB
  (R16)<31:0> ← tmp1 ! Release lock
  END
IF tmp1 EQ 0 THEN
  R0 ← 0 ! Queue was empty
ELSE
  BEGIN
    IF {tmp2 - R16} EQ 0 THEN
      R0 ← 2 ! Queue now empty
    ELSE
      R0 ← 1 ! Queue not empty
    END
  END
R1 ← addr ! Address of removed entry

```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Illegal Operand
- Translation Not Valid

Instruction mnemonics:

CALL_PAL REMQTIL

Remove from Longword Queue at Tail
Interlocked**Description:**

If the secondary interlock is clear, REMQTIL removes from the self-relative queue the entry preceding the header, pointed to by R16, and the address of the removed entry is returned in R1.

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal, a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

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 non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14).

10.3.16 Remove Entry from Longword Queue at Tail Interlocked Resident

Format:

CALL_PAL REMQTILR ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the queue was empty
!     1 if entry removed and queue still not empty
!     2 if entry removed and queue empty
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
! All parts of the Queue must be memory resident

N ← {retry_amount}                      ! Implementation-specific
REPEAT
  LOAD_LOCKED (tmp0 ← (R16))      ! Acquire hardware interlock.
  IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
    R0 ← -1, {return}              ! Already set
    done ← STORE_CONDITIONAL ((R16) ← {tmp0 OR 1} )
    N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

tmp1 ← SEXT(tmp0<31:0>)
tmp5 ← SEXT(tmp0<63:32>)
addr ← SEXT( {R16 + tmp5}<31:0> )
tmp2 ← SEXT( {addr + SEXT( (addr+4)<31:0>)}<31:0> )
(R16 + 4)<31:0> ← tmp2 - R16! Backward link of header
IF {tmp2 EQL R16} THEN
  (R16)<31:0> ← 0                      ! Forward link, release lock
ELSE
  BEGIN
    (tmp2)<31:0> ← R16 - tmp2 ! Forward link of predecessor
  MB
  (R16)<31:0> ← tmp1                      ! Release lock
  END
```

```

IF tmp1 EQ 0 THEN
  R0 ← 0                ! Queue was empty
ELSE
  IF {tmp2 - R16} EQ 0 THEN
    R0 ← 2              ! Queue now empty
  ELSE
    R0 ← 1              ! Queue not empty
  END
END
R1 ← addr                ! Address of removed entry

```

Exceptions:

Illegal Operand

Instruction mnemonics:

CALL_PAL REMQTLR Remove Entry from Longword Queue at Tail
Interlocked Resident

Description:

If the secondary interlock is clear, REMQTLR removes from the self-relative queue the entry preceding the header, pointed to by R16, and the address of the removed entry is returned in R1.

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal, a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

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 non-interruptible operation.

This instruction requires that the queue be memory resident and that the queue header and elements are quadword aligned. No alignment or memory management checks are made before starting queue modifications to verify these requirements. Therefore, if any of these requirements are not met, the queue may be left in an UNPREDICTABLE state and an illegal operand fault may be reported.

10.3.17 Remove Entry from Quadword Queue at Tail Interlocked

Format:

CALL_PAL REMQTIQ ! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the queue was empty
!     1 if entry removed and queue still not empty
!     2 if entry removed and queue empty
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
! Header and entries must be octaword aligned.
!
! Check header alignment
IF {R16<3:0> NE 0} THEN
    BEGIN
        {illegal operand exception}
    END

N ← {retry_amount}                      ! Implementation-specific
REPEAT
    LOAD_LOCKED (tmp1 ← (R16))        ! Acquire hardware interlock.
    IF tmp1<0> EQ 1 THEN                ! Try to set secondary interlock.
        R0 ← -1, {return}              ! Already set
        done ← STORE_CONDITIONAL ((R16) ← {tmp1 OR 1} )
        N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

tmp5 ← (R16+8)
IF tmp5<3:0> NE 0 THEN                ! Check Alignment
    BEGIN                              ! Release secondary interlock
        (R16) ← tmp1
        {illegal operand exception}
    END
! Check if the following can be done without
! causing a memory management exception:
! read contents of header + (header + 8) {if tmp1 NE 0}
! write into header + (header + 8)
! + (header + 8 + (header + 8)){if tmp1 NE 0}
```

```

IF {all memory accesses can NOT be completed} THEN
  BEGIN
    (R16) ← tmp1
    {initiate memory management fault}
  END

addr ← R16 + tmp5
tmp2 ← addr + (addr + 8)
IF tmp2<3:0> NE 0 THEN
  BEGIN
    (R16) ← tmp1
    {illegal operand exception}
  END

(R16 + 8) ← tmp2 - R16
IF {tmp2 EQL R16} THEN
  (R16) ← 0
ELSE
  BEGIN
    (tmp2) ← R16 - tmp2
    MB
    (R16) ← tmp1
  END
END
IF tmp1 EQ 0 THEN
  R0 ← 0
ELSE
  BEGIN
    IF {tmp2 - R16} EQ 0 THEN
      R0 ← 2
    ELSE
      R0 ← 1
    END
  END
END
R1 ← addr

```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Illegal Operand
- Translation Not Valid

Instruction mnemonics:

CALL_PAL REMQTIQ Remove from Quadword Queue at Tail Interlocked

Description:

If the secondary interlock is clear, REMQTIQ removes from the self-relative queue the entry preceding the header, pointed to by R16, and the address of the removed entry is returned in R1.

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal, a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

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 non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14).

10.3.18 Remove Entry from Quadword Queue at Tail Interlocked Resident

Format:

CALL_PAL

REMQTIQR

! PALcode format

Operation:

```
! R16 contains the address of the queue header
! R0 receives status:
!     -1 if the secondary interlock was set
!     0 if the queue was empty
!     1 if entry removed and queue still not empty
!     2 if entry removed and queue empty
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
! Header and entries must be octaword aligned.
! All parts of the Queue must be memory resident

N ← {retry_amount}           ! Implementation-specific
REPEAT
  LOAD_LOCKED (tmp1 ← (R16))  ! Acquire hardware interlock.
  IF tmp1<0> EQ 1 THEN        ! Try to set secondary interlock.
    R0 ← -1, {return}        ! Already set
    done ← STORE_CONDITIONAL ((R16) ← {tmp1 OR 1} )
    N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

tmp5 ← (R16+8)
addr ← R16 + tmp5
tmp2 ← addr + (addr + 8)
(R16 + 8) ← tmp2 - R16      ! Backward link of header
IF {tmp2 EQL R16} THEN
  (R16) ← 0                ! Forward link, release lock
ELSE
  BEGIN
    (tmp2) ← R16 - tmp2    ! Forward link of predecessor
    MB
    (R16) ← tmp1          ! Release lock
  END
END
```

```

IF tmp1 EQ 0 THEN
    R0 ← 0                                ! Queue was empty
ELSE
    IF {tmp2 - R16} EQ 0 THEN
        R0 ← 2                            ! Queue now empty
    ELSE
        R0 ← 1                            ! Queue not empty
    END
R1 ← addr                                ! Address of removed entry

```

Exceptions:

Illegal Operand

Instruction mnemonics:

CALL_PAL REMQTIQR Remove Entry from Quadword Queue at Tail
Interlocked Resident

Description:

If the secondary interlock is clear, REMQTIQR removes from the self-relative queue the entry preceding the header, pointed to by R16, and the address of the removed entry is returned in R1.

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal, a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R0 is set to -1. The value "N" is implementation dependent.

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 non-interruptible operation.

This instruction requires that the queue be memory resident and that the queue header and elements are octaword aligned. No alignment or memory management checks are made before starting queue modifications to verify these requirements. Therefore, if any of these requirements are not met, the queue may be left in an UNPREDICTABLE state and an illegal operand fault may be reported.

10.3.19 Remove Entry from Longword Queue

Format:

CALL_PAL REMQUEL ! PALcode format

Operation:

```
! R16 contains the address of the entry to remove
!   or the address of the 32 bit address of the
!   entry for REMQUEL/D
! R0 receives status:
!   -1 if the queue was empty
!   0 if the queue is empty after removing an entry
!   1 if the queue is not empty after removing an entry
! R1 receives the address of the removed entry
!
! Header and entries need only be byte aligned
! Must have write access to header and queue entries
IF opcode EQ REMQUEL/D THEN
    R1 ← SEXT((R16)<31:0>)
ELSE
    R1 ← SEXT(R16<31:0>)

IF {all memory accesses can be completed} THEN
    BEGIN
        tmp1 ← (R1)<31:0>                      ! Forward Link of Predecessor
        ((R1+4)<31:0><31:0> ← tmp1
        tmp2 ← (R1+4)<31:0>                      ! Backward Link of Successor
        ((R1)<31:0>+4)<31:0> ← tmp2
        R0 ← 1                                      ! Queue not empty
        IF {tmp1 EQ tmp2} THEN
            R0 ← 0                                      ! Queue now empty
        IF {R1 EQ tmp2} THEN
            R0 ← -1                                      ! Queue was empty
        END
    END
ELSE
    BEGIN
        {initiate fault}
    END
END
```

Exceptions:

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

Instruction mnemonics:

CALL_PAL	REMQUEL	Remove Entry from Longword Queue
CALL_PAL	REMQUEL/D	Remove Entry from Longword Queue Deferred

Description:

REMQUEL removes the entry addressed by R16 from the longword absolute queue. The address of the removed entry is returned in R1. REMQUEL/D performs the same operation on the queue entry addressed by the longword addressed by R16. The queue header and entry need only be byte aligned.

In either case, if there was no entry in the queue to be removed, R0 is set to -1. If there was an entry to remove and the queue is empty at the end of this instruction, R0 is set to 0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. The removal is a non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14).

10.3.20 Remove Entry from Quadword Queue

Format:

CALL_PAL REMQUEQ ! PALcode format

Operation:

```
! R16 contains the address of the entry to remove
!   or address of address of entry for REMQUEQ/D
! R0 receives status:
!   -1 if the queue was empty
!   0 if the queue is empty after removing an entry
!   1 if the queue is not empty after removing an entry
! R1 receives the address of the removed entry
! Must have write access to header and queue entries
! Header and entries must be octaword aligned
IF opcode EQ REMQUEQ/D THEN
  IF {R16<3:0> NE 0} THEN
    BEGIN
      {illegal operand exception}
    END
    R1 ← (R16)
  ELSE
    R1 ← R16
  IF {R1<3:0> NE 0} THEN        ! Check alignment
    BEGIN
      {illegal operand exception}
    END
  IF {all memory accesses can be completed} THEN
    BEGIN
      tmp1 ← (R1)                ! Forward link of Predecessor
      IF {tmp1<3:0> NE 0} THEN
        BEGIN                    ! Check alignment
          {illegal operand exception}
        END
      tmp2 ← (R1+8)              ! Find predecessor
      IF {tmp2<3:0> NE 0} THEN
        BEGIN                    ! Check alignment
          {illegal operand exception}
        END
      (tmp2) ← tmp1              ! Update Forward link of predecessor
      ((R1)+8) ← tmp2
```

```

R0 ← 1                ! Queue not empty
IF {tmp1 EQ tmp2} THEN
  R0 ← 0              ! Queue now empty
IF {R1 EQ tmp2} THEN
  R0 ← -1             ! Queue was empty
END
ELSE
  BEGIN
    {initiate fault}
  END
END

```

Exceptions:

- Access Violation
- Fault on Read
- Fault on Write
- Translation Not Valid
- Illegal Operand

Instruction mnemonics:

CALL_PAL	REMQUEQ	Remove Entry from Quadword Queue
CALL_PAL	REMQUEQ/D	Remove Entry from Quadword Queue Deferred

Description:

REMQUEQ removes the queue entry addressed by R16 from the quadword absolute queue. The address of the removed entry is returned in R1. REMQUEQ/D performs the same operation on the queue entry addressed by the quadword addressed by R16.

In either case, if there was no entry in the queue to be removed, R0 is set to -1. If there was an entry to remove and the queue is empty at the end of this instruction, R0 is set to 0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. The removal is a non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 11 and 14). R0 and R1 are UNPREDICTABLE if an exception occurs. The relative order of reporting memory management and illegal operand exceptions is UNPREDICTABLE.

10.4 Unprivileged VAX Compatibility PALcode Instructions

The Alpha architecture provides the following PALcode instructions 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 provided to help customers preserve VAX instruction atomicity assumptions in porting code from VAX to Alpha. These calls should be user mode. They must not be used by any code other than that generated by the VEST software translator and its supporting run-time code (TIE).

10.4.1 Atomic Move Operation

Format:

```
AMOVRR                                ! PALcode format
AMOVRM                                ! PALcode format
```

Operation:

```
! R16 contains the first source
! R17 contains the first destination address
! R18 contains the first length
! R19 contains the second source
! R20 contains the second destination address
! R21 contains the second length
CASE
  AMOVRR:
    IF intr_flag EQ 0 THEN
      R18 ← 0
      {return}
    END

    intr_flag ← 0
    (R17) ← R16      ! length specified by R18<1:0>
    (R20) ← R19      ! length specified by R21<1:0>
    IF {both moves successful} THEN
      R18 ← 1
    ELSE
      R18 ← 0
    END

  AMOVRM:
    IF intr_flag EQ 0 THEN
      R18 ← 0
      {return}
    END

    intr_flag ← 0
    (R17) ← R16      ! length specified by R18<1:0>
    IF R21<5:0> NE 0 THEN
      BEGIN
        IF R19<1:0> NE 0 OR R20<1:0> NE 0
          {Illegal operand exception}
        ELSE
          (R20) ← (R19)! length specified by R21<5:0>
        END
      END
    IF {both moves successful} THEN
      R18 ← 1
    ELSE
      R18 ← 0
    END
  END
ENDCASE
```

Exceptions:

AMOVRR:	Access Violation
	Fault On Write
	Translation Not Valid
AMOVRRM:	Access Violation
	Fault On Read
	Fault On Write
	Illegal Operand
	Translation Not Valid

Instruction mnemonics:

CALL_PAL	AMOVRR	Atomic Move Register/Register
CALL_PAL	AMOVRRM	Atomic Move Register/Memory

Description:**Note:**

The CALL_PAL AMOV_{xx} instructions exist *only* for the support of translated VAX code. They must be used *only* in translated VAX code and its support routines (TIE).

CALL_PAL AMOVRR

The CALL_PAL AMOVRR instruction specifies two multiprocessor-safe register stores to arbitrary byte addresses. Either both stores are done or neither store is done. R18 is set to 1 if both stores are done, and 0 otherwise. The two source registers are R16 and R19. The two destination byte addresses are in R17 and R20. The two lengths are specified in R18<1:0> and R21<1:0>. The length encoding is as follows: 00 is store byte, 01 is store word, 10 is store longword, 11 is store quadword. The low 1, 2, 4, or 8 bytes of the source register are used, respectively. The unused bytes of the source registers are ignored. The unused bits of the length registers (R18<63:2> and R21<63:2>) should be zero (SBZ).

If, upon entry to the PALcode routine, the intr_flag is clear then the instruction sets R18 to zero and exits, doing no stores. Otherwise, intr_flag is cleared and the PALcode routine proceeds. This is the same per-processor intr_flag used by the RS and RC instructions.

The AMOVRR memory addresses may be unaligned. If either store would result in a Translation Not Valid fault, Fault on Write, or Access Violation fault, neither store is done and the corresponding fault is taken. If both stores would result in faults, it is UNPREDICTABLE which one is taken.

Note:

A fault does not set R18, because the instruction has not been completed.

If both stores can be completed without faulting, they are both attempted using multiprocessor-safe LDQ_L..STQ_C sequences. If all the sequences store successfully with no interruption, the PALcode routine completes with R18 set to one. Otherwise, the PALcode routine completes with R18 set to zero. In addition, R16, R17, R19, R20, and R21 are UNPREDICTABLE upon return from the PALcode routine, even if an exception has occurred.

If the destinations overlap, the stores must appear to be done in the order specified.

CALL_PAL AMOVRM

The CALL_PAL AMOVRM instruction specifies one multiprocessor safe register store to an arbitrary byte address, plus an atomic memory-to-memory move of 0 to 63 aligned longwords. Either the store and the move are both done in their entirety or neither is done. R18 is set to one if both are done, and zero otherwise.

The first source register is R16, the first destination address is in R17, and the first length is in R18. These three are specified exactly as in AMOVRM.

The second source address is in R19, the second destination address is in R20, and the second length is in R21<5:0>. The length is a longword length, in the range 0 to 63 longwords (0 to 252 bytes). The unused bytes of the source register R16 are ignored. The unused bits of the length registers (R18<63:2> and R21<63:6>) should be zero (SBZ).

If, upon entry to the PALcode routine, the intr_flag is clear, the instruction sets R18 to zero and exits, doing no stores. Otherwise, intr_flag is cleared and the PALcode routine proceeds. This is the same per-processor intr_flag used by the RS and RC instructions.

The memory address in R17 may be unaligned.

If the length for the move is zero, no move is done, no memory accesses are made via R19 and R20, and no fault checking of these addresses is done. In this case, the move is always considered to have succeeded in determining the setting of R18.

If the length in R21 is non-zero, the two addresses in R19 and R20 must be aligned longword addresses; otherwise, an Illegal Operand exception is taken.

If either the store or the move would result in a Translation Not Valid, Fault on Read, Fault on Write, or Access Violation fault, neither is done and the corresponding fault is taken. If both would result in faults, it is UNPREDICTABLE which one is taken.

Note:

A fault does not set R18, since the instruction has not been completed.

If both the store and the move can be completed without faulting, they are both attempted, using multiprocessor-safe LDQ_L..STQ_C sequences for the store. If all the operations store successfully with no interruption, the PALcode routine completes with R18 set to one. Otherwise, the PALcode routine completes with R18 set to zero. In addition, R16, R17, R19, R20, and R21 are UNPREDICTABLE upon return from the PALcode routine, even if an exception has occurred.

If the memory fields overlap, the store must appear to be done first, followed by the move. The ordering of the reads and writes of the move is unspecified. Thus, if the move destination overlaps the move source, the move results are UNPREDICTABLE.

These instructions contain no implicit MB.

Notes:

- Typically, these instructions would be used in a sequence starting with CALL_PAL RS and ending with CALL_PAL AMOVxx, Bxx R18,label. The failure path from the conditional branch would eventually go back to the RS instruction. When such a sequence succeeds, it has done everything from the RS up to and including the CALL_PAL AMOVxx completely with no interrupts or exceptions.
- The CALL_PAL AMOVxx instruction is typically followed by a conditional branch on R18. If the CALL_PAL AMOVxx is likely to succeed, the conditional branch should be a forward branch on failure (BEQ R18,forward_label) or backward branch on success (BNE R18, backward_label), to match the architected branch-prediction rule.
- The CALL_PAL AMOVxx instruction must either do both stores or neither. If R18=0 upon return, then memory state must be unchanged. If the first STQ_C inside AMOVRR succeeds (and thus has changed programmer-visible state in memory), the PALcode routine must complete the second STQ_C also, and exit with R18=1. In particular, if the failure loop around the second STQ_C is executed an excessive number of times (due to perverse interference from another processor), the PALcode may not "give up" and return with R18=0.

10.5 Unprivileged PALcode Thread Instructions

The PALcode thread instructions provide support for multithread implementations, which require that a given thread be able to generate a reproducible unique value in a "timely" fashion. This value can then be used to index into a structure or otherwise generate additional thread unique data.

The two instructions in Table 10–4 are provided to read and write a process unique value from the process's hardware context.

Table 10–4: Unprivileged PALcode Thread Instructions

Mnemonic	Operation
READ_UNQ	Read unique context
WRITE_UNQ	Write unique context

The process-unique value is stored in the HWPCB at [HWPCB+72] when the process is not active. When the process is active, the process unique value can be cached in hardware internal storage or reside in the HWPCB only.

10.5.1 Read Unique Context

Format:

CALL_PAL READ_UNQ ! PALcode format

Operation:

```
IF {internal storage for process unique context} THEN
    R0 ← {process unique context}
ELSE
    R0 ← (HWPCB+72)
```

Exceptions:

None

Instruction mnemonics:

CALL_PAL READ_UNQ Read Unique Context

Description:

The READ_UNQ instruction causes the hardware process (thread) unique context value to be placed in R0. If this value has not previously been written using a CALL_PAL WRITE_UNQ or stored into the quadword in the HWPCB at [HWPCB+72] while the thread was inactive, the result returned in R0 is UNPREDICTABLE. Implementations can cache this unique context value while the hardware process is active. The unique context may be thought of as a "slow register." Typically, this value will be used by software to establish a unique context for a given thread of execution.

10.5.2 Write Unique Context

Format:

CALL_PAL WRITE_UNQ ! PALcode format

Operation:

!R16 contains value to be written to the hardware process
! unique context

```
IF {internal storage for process unique context} THEN
    {process unique context} ← R16
ELSE
    (HWPCB+72) ← R16
```

Exceptions:

None

Instruction mnemonics:

CALL_PAL WRITE_UNQ Write Unique Context

Description:

The WRITE_UNQ instruction causes the value of R16 to be stored in internal storage for hardware process (thread) unique context, if implemented, or in the HWPCB at [HWPCB+72], if the internal storage is not implemented. When the process is context switched, SWPCTX ensures that this value is stored in the HWPCB at [HWPCB+72]. Implementations can cache this unique context value in internal storage while the hardware process is active. The unique context may be thought of as a "slow register." Typically, this value will be used by software to establish a unique context for a given thread of execution.

10.6 Privileged PALcode Instructions

Privileged instructions can be called in kernel mode only; otherwise, a privileged instruction exception occurs. The following privileged instructions are provided:

Table 10–5: PALcode Privileged Instructions Summary

Mnemonic	Operation
CFLUSH	Cache flush
CSERVE	Console service
DRAINA	Drain abort. Section 6.7.1.
HALT	Halt processor. See Section 6.7.2.
LDQP	Load quadword physical
MFPR	Move from processor register
MTPR	Move to processor register
STQP	Store quadword physical
SWPCTX	Swap privileged context
SWPPAL	Swap PALcode image

10.6.1 Cache Flush

Format:

CALL_PAL CFLUSH ! PALcode format

Operation:

! R16 contains the Page Frame Number (PFN)
! of the page to be flushed

IF PS<CM> NE 0 THEN
 {privileged instruction exception}

{Flush page out of cache(s)}

Exceptions:

Privileged Instruction

Instruction mnemonics:

CALL_PAL CFLUSH Cache Flush

Description:

The CFLUSH instruction may be used to flush an entire physical page specified by the PFN in R16 from any data caches associated with the current processor. All processors must implement this instruction.

On processors that implement a backup power option that maintains only the contents of memory during a powerfail, this instruction is used by the powerfail interrupt handler to force data written by the handler to the battery backed-up main memory. After a CFLUSH, the first subsequent load (on the same processor) to an arbitrary address in the target page is either fetched from physical memory or from the data cache of another processor.

In some multiprocessor systems, CFLUSH is not sufficient to ensure that the data are actually written to memory and not exchanged between processor caches. Additional platform-specific cooperation between the powerfail interrupt handlers executing on each processor may be required.

On systems that implement other backup power options (including none), CFLUSH may return without affecting the data cache contents. To order CFLUSH properly with respect to preceding writes, an MB instruction is needed before the CFLUSH; to order CFLUSH properly with respect to subsequent reads, an MB instruction is needed after the CFLUSH.

10.6.2 Console Service

Format:

CALL_PAL CSERVE ! PALcode format

Operation:

```
! Implementation specific

IF PS<CM> NE 0 THEN
{Privileged instruction exception}

ELSE
{Implementation-dependent action}
```

Exceptions:

Privileged Instruction

Instruction mnemonics:

CALL_PAL CSERVE Console Service

Description:

This instruction is specific to each PALcode and console implementation and is not intended for operating system use.

10.6.3 Load Quadword Physical

Format:

CALL_PAL LDQP ! PALcode format

Operation:

! R16 contains the quadword-aligned physical address
! R0 receives the data from memory

IF PS<CM> NE 0 THEN
 {Privileged Instruction exception}

R0 ← (R16) {physical access}

Exceptions:

Privileged Instruction

Instruction mnemonics:

CALL_PAL LDQP Load Quadword Physical

Description:

The LDQP instruction fetches and writes to R0 the quadword-aligned memory operand, whose physical address is in R16.

If the operand address in R16 is not quadword aligned, the result is UNPREDICTABLE.

10.6.4 Move from Processor Register

Format:

CALL_PAL MFPR_IPR_Name ! PALcode format

Operation:

```
IF PS<CM> NE 0 THEN
    {privileged instruction exception}
```

```
! R16 may contain an IPR specific source operand
R0 ← result of IPR specific function
```

Exceptions:

Privileged Instruction

Instruction mnemonics:

CALL_PAL MFPR_*xxx* Move from Processor Register *xxx*

Description:

The MFPR_*xxx* instruction reads the internal processor register specified by the PALcode function field and writes it to R0.

Registers R1, R16, and R17 contain UNPREDICTABLE results after an MFPR.

See Chapter 13 for a description of each IPR.

10.6.5 Move to Processor Register

Format:

CALL_PAL MTPR_IPR_Name ! PALcode format

Operation:

IF PS<CM> NE 0 THEN
 {privileged instruction exception}
 ! R16 may contain an IPR specific source operand

R0 ← result of IPR specific function
IPR ← result of IPR specific function

Exceptions:

Privileged Instruction

Instruction mnemonics:

CALL_PAL MTPR_xxx Move to Processor Register xxx

Description:

The MTPR_xxx instruction writes the IPR-specific source operands in integer registers R16 and R17 (R17 reserved for future use) to the internal processor register specified by the PAL-code function field. The effect produced by loading a processor register is guaranteed to be active on the next instruction.

Registers R1, R16, and R17 contain UNPREDICTABLE results after an MTPR. The MTPR may return results in R0. If the specific IPR being accessed does not return results in R0, then R0 contains an UNPREDICTABLE result after an MTPR.

See Chapter 13 for a description of each IPR.

10.6.6 Store Quadword Physical

Format:

CALL_PAL STQP ! PALcode format

Operation:

! R16 contains the quadword aligned physical address
! R17 contains the data to be written

IF PS<CM> NE 0 then
 {Privileged Instruction exception}

(R16) ← R17 {physical access}

Exceptions:

Privileged Instruction

Instruction mnemonics:

CALL_PAL STQP Store Quadword Physical

Description:

The STQP instruction writes the quadword contents of R17 to the memory location whose physical address is in R16.

If the operand address in R16 is not quadword aligned, the result is UNPREDICTABLE.

10.6.7 Swap Privileged Context

Format:

```
CALL_PAL          SWPCTX          ! PALcode format
```

Operation:

```
! R16 contains the physical address of the new HWPCB.

! check HWPCB alignment

IF R16<6:0> NE 0 THEN
    {reserved operand exception}
IF {PS<CM> NE 0} THEN
    {privileged instruction exception}

! Store old HWPCB contents

(IPR_PCBB + HWPCB_KSP) ← SP
IF {internal registers for stack pointers} THEN
    BEGIN
        (IPR_PCBB + HWPCB_ESP) ← IPR_ESP
        (IPR_PCBB + HWPCB_SSP) ← IPR_SSP
        (IPR_PCBB + HWPCB_USP) ← IPR_USP
    END

IF {internal registers for ASTxx} THEN
    BEGIN
        (IPR_PCBB + HWPCB_ASTSR) ← IPR_ASTSR
        (IPR_PCBB + HWPCB_ASTEN) ← IPR_ASTEN
    END
tmp1 ← PCC
tmp2 ← ZEXT(tmp1<31:0>)
tmp3 ← ZEXT(tmp1<63:32>)
(IPR_PCBB + HWPCB_PCC) ← {tmp2 + tmp3}<31:0>
IF {internal storage for process unique value} THEN
    BEGIN
        (IPR_PCBB + HWPCB_UNQ) ← process unique value
    END

! Load new HWPCB contents

IPR_PCBB ← R16

IF {ASNs not implemented in virtual instruction cache} THEN
    {flush instruction cache}
```

```

IF {ASNs not implemented in TB} THEN
  IF {IPR_PTBR NE (IPR_PCBB + HWPCB_PTBR)} THEN
    {invalidate trans. buffer entries with PTE<ASM> EQ 0}
  ELSE
    IPR_ASN ← (IPR_PCBB + HWPCB_ASN)

SP ← (IPR_PCBB + HWPCB_KSP)
IF {internal registers for stack pointers} THEN
  BEGIN
    IPR_ESP ← (IPR_PCBB + HWPCB_ESP)
    IPR_SSP ← (IPR_PCBB + HWPCB_SSP)
    IPR_USP ← (IPR_PCBB + HWPCB_USP)
  END

IPR_PTBR ← (IPR_PCBB + HWPCB_PTBR)

IF {internal registers for ASTxx} THEN
  BEGIN
    IPR_ASTSR ← (IPR_PCBB + HWPCB_ASTSR)
    IPR_ASTEN ← (IPR_PCBB + HWPCB_ASTEN)
  END

IPR_FEN ← (IPR_PCBB + HWPCB_FEN)
tmp4 ← ZEXT((IPR_PCBB + HWPCB_PCC)<31:0>)
tmp4 ← tmp4 - tmp2
PCC<63:32> ← tmp4<31:0>
IF {internal storage for process unique value} THEN
  BEGIN
    process unique value ← (IPR_PCBB + HWPCB_UNQ)
  END
IF {internal storage for Data Alignment trap setting} THEN
  BEGIN
    DAT ← (IPR_PCBB + HWPCB_DAT)
  END

```

Exceptions:

- Reserved Operand
- Privileged Instruction

Instruction mnemonics:

CALL_PAL SWPCTX Swap Privileged Context

Description:

The SWPCTX instruction returns ownership of the current Hardware Privileged Context Block (HWPCB) to the operating system and passes ownership of the new HWPCB to the processor. The HWPCB is described in Section 12.2.

SWPCTX saves the privileged context from the internal processor registers into the HWPCB specified by the physical address in the PCBB internal processor register. It then loads the privileged context from the new HWPCB specified by the physical address in R16. The actual sequence of the save and restore operation is not specified, so any overlap of the current and new HWPCB storage areas produces UNDEFINED results.

The privileged context includes the four stack pointers, the Page Table Base Register (PTBR), the Address Space Number (ASN), the AST enable and summary registers, the Floating-point Enable Register (FEN), the Performance Monitor (PME) register, the Data Alignment Trap (DAT) register, and the Charged Process Cycles — the number of PCC register counts that are charged to a process (modulo 2^{32}).

PTBR is never saved in the HWPCB and it is UNPREDICTABLE whether or not ASN is saved. These values cannot be changed for a running process. The process integer and floating registers are saved and restored by the operating system. See Figure 12–1 for the HWPCB format.

Notes:

- Any change to the current HWPCB while the processor has ownership results in UNDEFINED operation.
- All the values in the current HWPCB can be read through IPRs, except the Charged Process Cycles.
- If the HWPCB is read while ownership resides with the processor, it is UNPREDICTABLE whether the original or an updated value of a field is read. The processor can update an HWPCB field at any time. The decision as to whether or not a field is updated is made individually for each field.
- If the enabling conditions are present for an interrupt at the completion of this instruction, the interrupt occurs before the next instruction.
- PALcode sets up the PCBB at boot time to point to the HWPCB storage area in the Hardware Restart Parameter Block (HWRPB). See Section 26.1.
- The operation is UNDEFINED if SWPCTX accesses a non-memory-like region.
- A reference to nonexistent memory causes a machine check. Unimplemented physical address bits are SBZ. The operation is UNDEFINED if any of these bits are set.

Note:

Processors may keep a copy of each of the per-process stack pointers in internal registers. In those processors, SWPCTX stores the internal registers into the HWPCB. Processors that do not keep a copy of the stack pointers in internal registers keep only the stack pointer for the current access mode in SP and switch this with the HWPCB contents whenever the current access mode changes.

10.6.8 Swap PALcode Image

Format:

```
CALL_PAL          SWPPAL          ! PALcode format
```

Operation:

```
! R16 contains the new PALcode identifier
! R17-R21 contain implementation-specific entry parameters
! R0  receives status:
!      0 Success (PALcode was switched)
!      1 Unknown PALcode variant
!      2 Known PALcode variant, but PALcode not loaded

IF (PS<CM> NE 0) then
    {Privileged instruction exception}

ELSE
    IF {R16 < 256} THEN
        BEGIN
            IF {R16 invalid} THEN
                R0 ← 1
                {Return}
            ELSE IF {PALcode not loaded} THEN
                R0 ← 2
                {Return}
            ELSE
                tmp1 ← {PALcode base}
        END
    ELSE
        tmp1 = R16
        {Flush instruction cache}
        {Invalidate all translation buffers}
        {Perform additional PALcode variant-specific initialization}
        {Transfer control to PALcode entry at physical address in tmp1}
```

Exceptions:

Privileged Instruction

Instruction mnemonics:

```
CALL_PAL  SWPPAL          Swap PALcode Image
```

Description:

The SWPPAL instruction causes the current (active) PALcode to be replaced by the specified new PALcode image. This instruction is intended for use by operating systems only during bootstraps and by consoles during transitions to console I/O mode.

The PALcode descriptor contained in R16 is interpreted as either a PALcode variant or the base physical address of the new PALcode image. If a variant, the PALcode image must have been previously loaded. No PALcode loading occurs as a result of this instruction.

After successful PALcode switching, the register contents are determined by the parameters passed in R17 through R21 or are UNPREDICTABLE. A common parameter is the address of a new HWPCB. In this case, the stack pointer register and PTBR are determined by the contents of that HWPCB; the contents of other registers such as R16 through R21 may be UNPREDICTABLE.

See Section 27.3.2, for information on using this instruction.

10.6.9 Wait for Interrupt

Format:

CALL_PAL WTINT ! PALcode format

Operation:

! R16 contains the maximum number of interval clock ticks to skip
! R0 receives the number of interval clock ticks actually skipped

```
IF (implemented)
BEGIN
  IF {Implementation supports skipping multiple
      clock interrupts} THEN
    {Ticks_to_skip ← R16}

    {Wait no longer than any non-clock interrupt or the first clock
     interrupt after ticks_to_skip ticks have been skipped}

  IF {Implementation supports skipping multiple}
    {clock interrupts} THEN
    R0 ← number of interval clock ticks actually skipped
  ELSE
    R0 ← 0
END
ELSE
  R0 ← 0
{return}
```

Exceptions:

Privileged Instruction

Instruction mnemonics:

CALL_PAL WTINT Wait for Interrupt

Description:

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
- The first clock tick after a specified number of clock ticks has been skipped

The WTINT instruction returns in R0 the number of clock ticks that are skipped. The number returned in R0 is zero on hardware platforms that implement this instruction, but where it is not possible to skip clock ticks.

The operating system can specify a full 64-bit integer value in R16 as the maximum number of interval clock ticks to skip. A value of zero in R16 causes no clock ticks to be skipped.

Note the following if specifying in R16 the maximum number of interval clock ticks to skip:

- Adherence to a specified value in R16 is at the discretion of the PALcode; the PALcode may complete execution of WTINT and proceed to the next instruction at any time up to the specified maximum, even if no interrupt or interval-clock tick has occurred. That is, WTINT may return before all requested clock ticks are skipped.
- The PALcode must complete execution of WTINT if an interrupt occurs or if an interval-clock tick occurs after the requested number of interval-clock ticks has been skipped.

In a multiprocessor environment, only the issuing processor is affected by an issued WTINT instruction. The counters, SCC and PCC, may increment at a lower rate or may stop entirely during WTINT execution. This side effect is implementation dependent.

Memory Management (II-A)

11.1 Introduction

Memory management consists of the hardware and software that control the allocation and use of physical memory. Typically, in a multiprogramming system, several processes may reside in physical memory at the same time (see Chapter 12). OpenVMS uses memory protection and multiple address spaces to ensure that one process will not affect other processes or the operating system.

To further improve software reliability, four hierarchical access modes provide memory access control. They are, from most to least privileged: kernel, executive, supervisor, and user. Protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write for each of the four access modes. Accessible pages can be restricted to have only data or instruction access.

A program uses virtual addresses to access its data and instructions. However, before these virtual addresses can be used to access memory, they must be translated into physical addresses. Memory management software maintains hierarchical tables of mapping information (page tables) that keep track of where each virtual page is located in physical memory. The processor utilizes this mapping information when it translates virtual addresses to physical addresses.

Therefore, memory management provides mechanisms for both memory protection and memory mapping. The OpenVMS memory management architecture is designed to meet several goals:

- Provide a large address space for instructions and data
- Allow programs to run on hardware with physical memory smaller than the virtual memory used
- Provide convenient and efficient sharing of instructions and data
- Allow sparse use of a large address space without excessive page table overhead
- Contribute to software reliability
- Provide independent read and write access protection

11.2 Virtual Address Space

A virtual address is a 64-bit unsigned integer that specifies a byte location within the virtual address space. Implementations subset the address space supported to one of several sizes, as a function of page size and page table depth. The minimal virtual address size supported is 43 bits. If an implementation supports less than 64-bit virtual addresses, it must check that all the

VA<63:VA_SIZE> bits are equal to VA<VA_SIZE-1>. That gives two disjoint ranges for valid virtual addresses. For example, for a 43-bit virtual address space, valid virtual address ranges are 0...3FF FFFF FFFF₁₆ and FFFF FC00 0000 0000₁₆...FFFF FFFF FFFF FFFF₁₆. Accesses to virtual addresses outside of the valid virtual address ranges for an implementation cause an access violation exception.

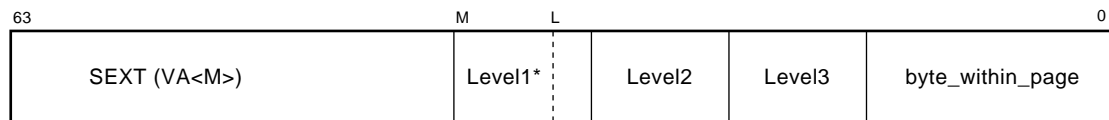
The virtual address space is broken into pages, which are the units of relocation, sharing, and protection. The page size ranges from 8K bytes to 64K bytes. System software should, therefore, allocate regions with differing protection on 64K-byte virtual address boundaries to ensure image compatibility across all Alpha implementations.

Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. The operating system controls the virtual-to-physical address mapping tables and saves the inactive parts of the virtual address space on external storage media.

11.3 Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. The virtual address consists of three level-number fields and a byte_within_page field, as shown in Figure 11-1.

Figure 11-1: Virtual Address Format



* Level1 <M:L+1> contains SEXT(VA<L>), where L is the highest numbered implemented VA bit.

The byte_within_page field can be either 13, 14, 15, or 16 bits depending on a particular implementation. Thus, the allowable page sizes are 8K bytes, 16K bytes, 32K bytes, and 64K bytes. Each level-number field contains *n* bits, where *n* is, for example, 10 with an 8K-byte page size.

An implementation may support a smaller virtual address space than the page size allows by including only a subset of low-order bits in Level1. The smaller virtual address space must be at least 43 bits and must be large enough that at least two bits of Level1 are implemented.¹

¹ OpenVMS requires at least three PTEs in the highest-level page table. The lowest-order PTE must map process space, the highest-order PTE must map system space and another PTE maps the page table structure. See Section 11.8.2.

The level-number fields are a function of the page size; all page table entries at any given level do not exceed one page. The PFN field in the PTE is always 32 bits wide. Thus, as the page size grows, the virtual and physical address size also grows (Table 11–1).

Table 11–1 Virtual Address Options

Page Size (bytes)	Byte Offset (bits)	Level Size (bits)	Virtual Address (bits)	Physical Address (bits)
8 K	13	10	43	45
16 K	14	11	43–47 [†]	46
32 K	15	12	43–51 [†]	47
64 K	16	13	44–55 [†]	48

[†] Level1 page table might be partially utilized for this page size.

11.4 Physical Address Space

Physical addresses are at most 48 bits. A processor may choose to implement a smaller physical address space by not implementing some number of high-order bits.

The two most significant implemented physical address bits delineate the four regions in the physical address space. Implementations use these bits as appropriate for their systems. For example, in a workstation with a 30-bit physical address space, bit <29> might select between memory and non-memory-like regions, and bit <28> could enable or disable caching. See Chapter 5.

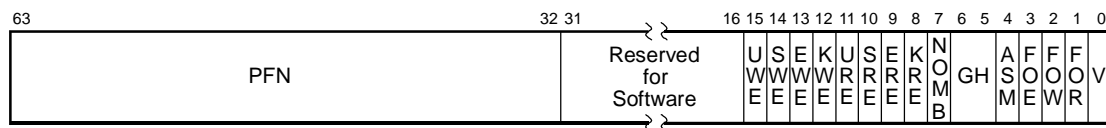
11.5 Memory Management Control

Memory management is always enabled. Implementations must provide an environment for PALcode to service exceptions and to initialize and boot the processor. For example, PALcode might run with I-stream mapping disabled and use the privileged CALL_PAL LDQP and STQP instructions to access data stored in physical addresses.

11.6 Page Table Entries

The processor uses a quadword Page Table Entry (PTE), as shown in Figure 11–2, to translate virtual addresses to physical addresses. A PTE contains hardware and software control information and the physical Page Frame Number.

Figure 11–2 Page Table Entry



Fields in the page table entry are interpreted as shown in Table 11–2.

Table 11–2 Page Table Entry

Bits	Description
63–32	Page Frame Number (PFN) The PFN field always points to a page boundary. If V is set, the PFN is concatenated with the byte_within_page bits of the virtual address to obtain the physical address (see Section 11.8). If V is clear, this field may be used by software.
31–16	Reserved for software.
15	User Write Enable (UWE) This bit enables writes from user mode. If this bit is a 0 and a STORE is attempted while in user mode, an Access Violation occurs. This bit is valid even when V=0. Note: If a write-enable bit is set and the corresponding read-enable bit is not, the operation of the processor is UNDEFINED.
14	Supervisor Write Enable (SWE) This bit enables writes from supervisor mode. If this bit is a 0 and a STORE is attempted while in supervisor mode, an Access Violation occurs. This bit is valid even when V=0.
13	Executive Write Enable (EWE) This bit enables writes from executive mode. If this bit is a 0 and a STORE is attempted while in executive mode, an Access Violation occurs. This bit is valid even when V=0.
12	Kernel Write Enable (KWE) This bit enables writes from kernel mode. If this bit is a 0 and a STORE is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V=0.
11	User Read Enable (URE) This bit enables reads from user mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in user mode, an Access Violation occurs. This bit is valid even when V=0.
10	Supervisor Read Enable (SRE) This bit enables reads from supervisor mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in supervisor mode, an Access Violation occurs. This bit is valid even when V=0.
9	Executive Read Enable (ERE) This bit enables reads from executive mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in executive mode, an Access Violation occurs. This bit is valid even when V=0.
8	Kernel Read Enable (KRE) This bit enables reads from kernel mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V=0.

Table 11–2 Page Table Entry (Continued)

Bits	Description																																		
7	<p>Translation Buffer Miss Memory Barrier (NOMB)</p> <p>When set, the requirement described in Section 5.6.4.3 is lifted for ensuring that all processors using a newly valid PTE also see any new contents of the related page. This allows the TB-miss code to avoid potentially expensive global synchronization. Software is expected to set this bit on PTEs when it is known that the page contents are already visible to all processors.</p>																																		
6–5	<p>Granularity hint (GH)</p> <p>Software may set these bits as follows to supply a hint to translation buffer implementations that a block of pages can be treated as a single larger page:</p> <table border="1"> <thead> <tr> <th rowspan="2">PTE<6:5></th> <th colspan="4">Page Size Before GH:</th> </tr> <tr> <th>8KB</th> <th>16KB</th> <th>32KB</th> <th>64KB</th> </tr> </thead> <tbody> <tr> <td></td> <td colspan="4">Resulting Page Size:</td> </tr> <tr> <td>00</td> <td>8KB</td> <td>1 KB</td> <td>32KB</td> <td>64KB</td> </tr> <tr> <td>01</td> <td>64KB</td> <td>128KB</td> <td>256KB</td> <td>2MB</td> </tr> <tr> <td>10</td> <td>512KB</td> <td>1MB</td> <td>2MB</td> <td>64MB</td> </tr> <tr> <td>11</td> <td>4MB</td> <td>8MB</td> <td>16MB</td> <td>512MB</td> </tr> </tbody> </table>	PTE<6:5>	Page Size Before GH:				8KB	16KB	32KB	64KB		Resulting Page Size:				00	8KB	1 KB	32KB	64KB	01	64KB	128KB	256KB	2MB	10	512KB	1MB	2MB	64MB	11	4MB	8MB	16MB	512MB
PTE<6:5>	Page Size Before GH:																																		
	8KB	16KB	32KB	64KB																															
	Resulting Page Size:																																		
00	8KB	1 KB	32KB	64KB																															
01	64KB	128KB	256KB	2MB																															
10	512KB	1MB	2MB	64MB																															
11	4MB	8MB	16MB	512MB																															
	<p>Note:</p> <ol style="list-style-type: none"> 1. The block is a group of physically contiguous pages that are naturally aligned both virtually and physically. Within the block, the PFN field in each PTE must map the correct physical page for the virtual page to which the PTE corresponds. 2. Within the block, all PTEs have the same values for bits <15:0>, that is, protection, fault, granularity, and valid bits. <p>Hardware may use this hint to map the entire block with a single TB entry.</p> <p>It is UNPREDICTABLE which PTE values within the block are used if the granularity bits are set inconsistently.</p> <p>Programming Note:</p> <p>A granularity hint might be appropriate for a large memory structure such as a frame buffer or nonpaged pool that, in fact, is mapped into contiguous virtual pages with identical protection, fault, and valid bits.</p>																																		
4	<p>Address Space Match (ASM)</p> <p>When set, this PTE matches all Address Space Numbers. For a given VA, ASM must be set consistently in all processes; otherwise, the address mapping is UNPREDICTABLE.</p>																																		
3	<p>Fault on Execute (FOE)</p> <p>When set, a Fault on Execute exception occurs on an attempt to execute an instruction in the page.</p>																																		

Table 11–2 Page Table Entry (Continued)

Bits	Description
2	Fault on Write (FOW) When set, a Fault on Write exception occurs on an attempt to write any location in the page.
1	Fault on Read (FOR) When set, a Fault on Read exception occurs on an attempt to read any location in the page.
0	Valid (V) Indicates the validity of the PFN field. When V is set, the PFN field is valid for use by hardware. When V is clear, the PFN field is reserved for use by software. The V bit does not affect the validity of PTE<15:1> bits.

11.6.1 Changes to Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, the operating system may set or clear the valid bit, change the PFN field as pages are moved to and from external storage media, or modify the software bits. The processor hardware never changes PTEs.

Software must guarantee that each PTE is always internally consistent. Changing a PTE one field at a time may give incorrect system operation, for example, setting PTE<V> with one instruction before establishing PTE<PFN> with another. Execution of an interrupt service routine between the two instructions could use an address that would map using the inconsistent PTE. Software can solve this problem by building a complete new PTE in a register and then moving the new PTE to the page table using a Store Quadword instruction (STQ).

Multiprocessing complicates the problem. Another processor could be reading (or even changing) the same PTE that the first processor is changing. Such concurrent access must produce consistent results. Software must use some form of software synchronization to modify PTEs that are already valid. Once a processor has modified a valid PTE, it is possible that other processors in a multiprocessor system may have old copies of that PTE in their Translation Buffer. When software changes a PTE, each processor may use either the old or the new PTE until software performs a TB invalidate on that processor (after which, the processor may use only the new PTE). An example of a case where either the old or new PTE could usefully be used is when the PTE<NOMB> bit is transitioned from zero to one.

Software may write new values into invalid PTEs using quadword store instructions (STQ). Hardware must ensure that aligned quadword reads and writes are atomic operations. The following procedure must be used to change any of the PTE bits <15:0> of a shared valid PTE (PTE<0>=1) such that an access that was allowed before the change is not allowed after the change.

1. The PTE<0> is cleared without changing any of the PTE bits <63:32> and <15:1>.
2. All processors do a TBIS for the VA mapped by the PTE that changed. The VA used in the TBIS must assume that the PTE granularity hint bits are zero.
3. After all processors have done the TBIS, the new PTE may be written changing any or all fields.

Programming Note:

The procedure above allows queue instructions that have probed in order to check that all can complete, to service a TB miss. The queue instructions use the PTE even though the V bit is clear, if the V bit was set during the instruction's initial probe flow.

11.7 Memory Protection

Memory protection is the function of validating whether a particular type of access is allowed to a specific page from a particular access mode. Access to each page is controlled by a protection code that specifies, for each access mode, whether read or write references are allowed.

The processor uses the following to determine whether an intended access is allowed:

- The virtual address, which is used to index page tables
- The intended access type (read data, write data, or instruction fetch)
- The current access mode from the Processor Status

If the access is allowed and the address can be mapped (the Page Table Entry is valid), the result is the physical address that corresponds to the specified virtual address.

For protection checks, the intended access is read for data loads and instruction fetch, and write for data stores.

If an operand is an address operand, then no reference is made to memory. Hence, the page need not be accessible nor map to a physical page.

11.7.1 Processor Access Modes

There are four processor modes:

- Kernel
- Executive
- Supervisor
- User

The access mode of a running process is stored in the Current Mode bits of the Processor Status (PS) (see Section 14–2).

11.7.2 Protection Code

Every page in the virtual address space is protected according to its use. A program may be prevented from reading or writing portions of its address space. Each page has an associated protection code that describes the accessibility of the page for each processor mode. The code allows a choice of read or write protection for each processor mode.

- Each mode's access can be read/write, read-only, or no-access.
- Read and write accessibility are specified independently.
- The protection of each mode can be specified independently.

The protection code is specified by 8 bits in the PTE (see Table 11–2).

The OpenVMS architecture allows a page to be designated as execute only by setting the read enable bit for the access mode and by setting the fault on read and write bits in the PTE.

11.7.3 Access Violation Fault

An Access Violation fault occurs if an illegal access is attempted, as determined by the current processor mode and the page's protection field.

11.8 Address Translation

The page tables can be accessed from physical memory, or (to reduce overhead) through a self-mapping to a linear region of the virtual address space. All implementations must support the virtual access method and are expected to use it as the primary access method to enhance performance.

Additionally, an optional reduced page table (RPT) mode is defined, which allows more efficient mapping of very large blocks of memory.

The following sections describe the access methods.

11.8.1 Physical Access for Page Table Entries

Physical address translation is performed by accessing entries in a multilevel page table structure. The Page Table Base Register (PTBR) contains the physical Page Frame Number (PFN) of the highest-level page table.

In systems that implement the Virtual Address Boundary (VIRBND) register, the System Page Table Base Register (SYSPTBR) contains the PFN of an alternate highest-level page table. In such systems, the virtual address to be translated is compared against the address stored in VIRBND. Translations of lower addresses begin with the PFN in PTBR as the highest-level page table. Translations of higher or equal addresses use the PFN in SYSPTBR as the highest-level page table. The VIRBND and SYSPTBR registers are described in Sections 13.3.24 and 13.3.18, respectively.

Level1 is the highest-level page table. Bits <Level1> of the virtual address are used to index into the Level1 page table to obtain the physical PFN of the base of the next level (Level2) page table. Bits <Level2> of the virtual address are used to index into the Level2 page table to obtain the physical PFN of the base of the next level (Level3) page table. Bits <Level3> of the virtual address are used to index into the Level3 page table to obtain the physical PFN of the page being referenced. The PFN is concatenated with virtual address bits <byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table resides in I/O space, or in nonexistent memory, the operation of the processor is UNDEFINED.

If all the higher-level PTEs (those PTEs that map higher-significance portions of the virtual address space than is mapped by Level3) are valid, the protection bits are ignored; the protection code in the Level3 PTE is used to determine accessibility. If a higher-level PTE is invalid, an access-violation fault occurs if the PTE<KRE> equals zero. An Access-Violation fault on any higher-level PTE implies that all lower-level page tables mapped by that PTE do not exist.

Programming Note:

This mapping scheme does not require multiple contiguous physical pages. There are no length registers. With a page size of 8KB, 3 pages (24KB) map 8MB of virtual address space; 1026 pages (approximately 8MB) map an 8GB address space; and 1,049,601 pages (approximately 8GB) map the entire 8TB 2^{43} byte address space.

The algorithm to generate a physical address from a virtual address follows:

```
IF {SEXT(VA<63:VA_SIZE>) NEQ SEXT(VA<VA_SIZE-1>)} THEN
    {initiate Access Violation fault}

IF (VIRBND in use) THEN
    IF (VA LTU VIRBND) THEN
        ptbr_value <- PTBR
    ELSE
        ptbr_value <- SYSPTBR
ELSE
    ptbr_value <- PTBR

! Read Physical
level1_pte <- ( { ptbr_value * page_size } + { 8 * VA<level1} )

IF level1_pte<V> EQ 0 THEN
    IF level1_pte<KRE> EQ 0 THEN
        {initiate Access Violation fault}
    ELSE
        {initiate Translation Not Valid fault}

! Read Physical
level2_pte <- ( {level1_pte<PFN> * page_size } + {8 * VA<level2>} )

IF level2_pte<V> EQ 0 THEN
    IF level2_pte<KRE> EQ 0 THEN
        {initiate Access Violation fault}
    ELSE
        {initiate Translation Not Valid fault}

! Read Physical
level3_pte <- ( {level2_pte<PFN> * page_size } + {8 * VA<level3>} )

IF {{{level3_pte<UWE> EQ 0} AND {write access} AND {PS<CM> EQ 3}} OR
    {{{level3_pte<URE> EQ 0} AND {read access} AND {PS<CM> EQ 3}} OR
    {{{level3_pte<SWE> EQ 0} AND {write access} AND {PS<CM> EQ 2}} OR
    {{{level3_pte<SRE> EQ 0} AND {read access} AND {PS<CM> EQ 2}} OR
    {{{level3_pte<EWE> EQ 0} AND {write access} AND {PS<CM> EQ 1}} OR
    {{{level3_pte<ERE> EQ 0} AND {read access} AND {PS<CM> EQ 1}} OR
    {{{level3_pte<KWE> EQ 0} AND {write access} AND {PS<CM> EQ 0}} OR
    {{{level3_pte<KRE> EQ 0} AND {read access} AND {PS<CM> EQ 0}}}
THEN
    {initiate Access Violation fault}
ELSE
```

```

    IF level3_pte<V> EQ 0 THEN
        {initiate Translation Not Valid fault}
    IF {level3_pte<FOW> EQ 1} AND {write access} THEN
        {initiate Fault On Write fault}
    IF {level3_pte<FOR> EQ 1} AND {read access} THEN
        {initiate Fault On Read fault}
    IF {level3_pte<FOE> EQ 1} AND {execute access} THEN
        {initiate Fault On Execute fault}

    Physical_Address ← {level3_pte<PFN> * page_size} OR VA<byte_within_page>

```

11.8.2 Virtual Access for Page Table Entries

To reduce the overhead associated with the address translation in a multilevel page table structure, the page tables are mapped into a linear region of the virtual address space. The virtual address of the base of the page table structure is set on a system-wide basis and is contained in the VPTB IPR.

When a native mode DTB or ITB miss occurs, the TBMISSE flows attempt to load the Level3 page table entry using a single virtual mode load instruction.

The algorithm involving the manipulation of the missing VA follows, where *pS* represents *pageSize*:

```

    tmp ← LEFT_SHIFT (va, {64 - {{lg(pS) * 4} - 9 }})
    tmp ← RIGHT_SHIFT (tmp, {64 - {{lg(pS) * 4} - 9 } + lg(pS)-3})
    tmp ← VPTB OR tmp
    tmp<2:0> ← 0

```

At this point, *tmp* contains the VA of the Level3 page table entry. A LDQ from that VA will result in the acquisition of the PTE needed to satisfy the initial TBMISSE condition.

However, in the PALcode environment, if a TBMISSE occurs during an attempt to fetch the Level3 PTE, it is necessary to use the longer sequence of multiple dependent loads described in Section 11.8.1.

Section 13.3.25 contains the description of the VPTB IPR used to contain the virtual address of the base of the page table structure.

The necessary mapping of the page tables for the correct function of the algorithm is done as follows.

1. Select a $2^{(3*\lg(\text{pageSize}/8))+3}$ byte-aligned region (an address with $3*\lg(\text{pageSize}/8)+3$ low-order zeros) in the virtual address space. This value will be written into the VPTB register.
2. Create one or two PTEs to map the page tables. Only one is required unless SYSPTBR is implemented and software intends to use it (that is, VIRBND is to be set to a value other than -1). Each PTE is initialized as follows:

```

PTE = 0                                ! Initialize all fields to zero
PTE<63:32> = PFN of Level1 pagetable    ! Set to the PFN from either the
                                         ! PTBR or SYSPTBR
PTE<8> = 1                              ! Set the kernel read enable bit
PTE<0> = 1                              ! Set the valid bit

```

3. Write the resulting PTE(s) into the page table entries that correspond to the VPTB value. The PTE that contains the PTBR's PFN is written to the page indicated by PTBR. If SYSPTBR is in use, the PTE that contains the SYSPTBR's PFN is written to the page indicated by SYSPTBR.

In either case, these are the Level1 page tables.

4. Set all Level1 and Level2 Valid PTEs to allow kernel read access.
5. Write the VPTB register with the selected base value.

Notes:

No validity checks need be made on the value stored in the VPTB in a running system. Therefore, if the VPTB contains an invalid address, the operation is UNDEFINED.

SYSPTBR allows software to replicate portions of virtual memory contents in physical memory. For example, in systems exhibiting non-uniform memory access times, read-only portions of the operating system may be separately instantiated in portions of physical memory, which provides the fastest access time to a given processor. An identical virtual address reference executed by multiple processors would translate by using each respective processor's SYSPTBR to the physical memory instance that is local to that processor, thereby increasing performance.

The physical page tables indicated by PTBR and SYSPTBR together map a single 64-bit virtual address space. They also map themselves into a single linear region of the address space, presenting to software one virtually accessible page table that maps the entire address space. The set of Level3 PTEs contributed by each physical page table are essentially disjoint from each other, with only the set indicated by PTBR being context-switched.

11.8.3 Reduced Page Table (RPT) Mode

The reduced page table (RPT) mode is an optional extension of 64KB page size mode. A portion of the address space is mapped by one fewer page table levels, allowing each of the entries in the lowest-level page table to map a 512MB page. In implementations that support granularity hints in hardware, applications can use these hints to make more efficient use of the translation buffer. Applications that can use the 512MB granularity hint in 64KB page size mode can use RPT mode for additional benefits.

With the 512MB granularity hint but without RPT, every entry in the Level3 page table maps the same 512MB page. With RPT, that Level3 page table is eliminated entirely, and the Level2 PTE that would normally point to that Level3 page table is used to directly map the 512MB page.

Therefore, in an RPT region, there is elimination of redundant page table pages and compression of page table space. The compressed PTEs are more likely to fit in hardware caches. If there is locality of reference, a new PTE that is needed to satisfy a mapping is more likely to be present in the cache. Additionally, a single TB entry that maps the VA of the lowest-level page table now allows access to PTEs mapping 4 TB, rather than 512 MB, of memory.

In order to use RPT mode, the feature must be available and enabled in the implementation, and:

- Use the 64KB page size.

- Every L2 PTE in the reduced page table region must have PTE<GH>=11₂, that is, a 512MB page size.
- The PFN field of the PTE must refer to a 512MB aligned page.
- The RPT region is selected by using VAs such that VA<vaSize-1:vaSize-2>=01₂.

11.8.3.1 Physical Access for Page Table Entries in Reduced Page Table Mode

Physical address translation is performed by accessing entries in a two-level page table structure. The Page Table Base Register (PTBR) contains the physical Page Frame Number (PFN) of the highest-level (Level1) page table.

In systems that implement the Virtual Address Boundary register (VIRBND), the System Page Table Base Register (SYSPTBR) contains the PFN of an alternate highest-level page table. In such systems, the virtual address to be translated is compared against the address stored in VIRBND. Translations of Level2 addresses begin with the PFN in PTBR as the highest-level page table. Translations of Level1 addresses use the PFN in SYSPTBR as the highest-level page table. The VIRBND and SYSPTBR registers are described in Sections 13.3.24 and 13.3.18, respectively.

Level1 is the highest-level page table. Bits <Level1> of the virtual address are used to index into the Level1 page table to obtain the physical PFN of the base of the next level (Level2) page table. Bits <Level2> of the virtual address are used to index into the Level2 page table to obtain the physical PFN of the page being referenced. The PFN is concatenated with virtual address bits <byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table resides in I/O space, or in nonexistent memory, the operation of the processor is UNDEFINED.

If the Level1 PTE is valid, the protection bits are ignored; the protection code in the Level2 PTE is used to determine accessibility. If a Level1 PTE is invalid, an access-violation fault occurs if the PTE<KRE> equals zero. An access-violation fault on any Level1 PTE implies that all Level2 page tables mapped by that PTE do not exist.

The algorithm to generate a physical address from a virtual address follows:

```

IF {SEXT(VA<63:VA_SIZE>) NEQ SEXT(VA<VA_SIZE-1>)} THEN
  {initiate Access Violation fault}
IF (VIRBND in use) THEN
  IF (VA LTU VIRBND) THEN
    ptbr_value <- PTRB
  ELSE
    ptbr_value <- SYSPTBR
ELSE
  ptbr_value <- PTBR

! Read Physical
level1_pte <-( { ptbr_value * page_size} + { 8 * VA<level1} )

```

```

IF level1_pte<V> EQ 0 THEN
  IF level1_pte<KRE> EQ 0 THEN
    {initiate Access Violation fault}
  ELSE
    {initiate Translation Not Valid fault}

! Read Physical
level2_pte ← ({level1_pte<PFN> * page_size} + {8 * VA<level2>})

IF {{{level2_pte<UWE> EQ 0} AND {write access} AND {PS<CM> EQ 3}} OR
   {{level2_pte<URE> EQ 0} AND {read access} AND {PS<CM> EQ 3}} OR
   {{level2_pte<SWE> EQ 0} AND {write access} AND {PS<CM> EQ 2}} OR
   {{level2_pte<SRE> EQ 0} AND {read access} AND {PS<CM> EQ 2}} OR
   {{level2_pte<EWE> EQ 0} AND {write access} AND {PS<CM> EQ 1}} OR
   {{level2_pte<ERE> EQ 0} AND {read access} AND {PS<CM> EQ 1}} OR
   {{level2_pte<KWE> EQ 0} AND {write access} AND {PS<CM> EQ 0}} OR
   {{level2_pte<KRE> EQ 0} AND {read access} AND {PS<CM> EQ 0}}}
THEN
  {initiate Access Violation fault}
ELSE
  IF level2_pte<V> EQ 0 THEN
    {initiate Translation Not Valid fault}
  IF {level2_pte<FOW> EQ 1} AND {write access} THEN
    {initiate Fault On Write fault}
  IF {level2_pte<FOR> EQ 1} AND {read access} THEN
    {initiate Fault On Read fault}
  IF {level2_pte<FOE> EQ 1} AND {execute access} THEN
    {initiate Fault On Execute fault}

Physical_Address ← {level2_pte<PFN> * page_size} OR VA<byte_within_RPT_page1>

```

11.8.3.2 Virtual Access for Page Table Entries in Reduced Page Table Mode

To reduce overhead associated with the address translation in a multilevel page table structure, the page tables are mapped into a linear region of the virtual address space. The virtual address of the base of the page table structure is set on a system-wide basis and is contained in the VPTB IPR.

When a native mode DTB or ITB miss occurs, it is desirable that the TBMISSE flow attempt to load the lowest-level PTE by using a single virtual load instruction without regard to whether the missing VA is mapped by two levels (RPT) or three levels of page table. (See Section E.2.2 for the 21364 implementation.)

11.9 Translation Buffer

In order to save actual memory references when repeatedly referencing the same pages, hardware implementations include a translation buffer to remember successful virtual address translations and page states.

¹ byte_within_RPT_page contains those bits that would have been VA<Level3>, concatenated with the VA<byte_within_page> field for 64 KB page table mode.

When the process context is changed, a new value is loaded into the Address Space Number (ASN) internal processor register with a Swap Privileged Context instruction (CALL_PAL SWPCTX). (See Section 10.6 and Chapter 12.) This causes address translations for pages with PTE<ASM> clear to be invalidated on a processor that does not implement address space numbers. Additionally, when the software changes any part (except for the Software field) of a valid Page Table Entry, it must also move a virtual address within the corresponding page to the Translation Buffer Invalidate Single (TBIS) internal processor register with the MTPR instruction (see Section 13.3.22). Changes to PTE<NOMB> are also an exception to this requirement. This bit only has an effect when a PTE is loaded into the translation buffer. Thus, there is no need to invalidate the TB when the bit changes.

Implementation Note:

Some implementations may invalidate the entire Translation Buffer on an MTPR to TBIS. In general, implementations may invalidate more than the required translations in the TB.

The entire Translation Buffer can be invalidated by doing a write to Translation Buffer Invalidate All register (CALL_PAL MTPR_TBIA), and all ASM=0 entries can be invalidated by doing a write to Translation Buffer Invalidate All Process register (CALL_PAL MTPR_TBIAP). See Section 13.3.21.

The Translation Buffer must not store invalid PTEs. Therefore, the software is not required to invalidate Translation Buffer entries when making changes for PTEs that are already invalid.

After software changes a valid Level1 or Level2 PTE, software must flush the translation for the corresponding page in the virtual page table. Then software must flush the translations of all valid pages mapped by that page. In the case of a change to a Level1 PTE, this action must be taken through a second iteration.

The TBCHK internal processor register is available for interrogating the presence of a valid translation in the Translation Buffer (see Section 13.3.19).

Implementation Note:

Hardware implementors should be aware that a single, direct-mapped TB has a potential problem when a load/store instruction and its data map to the same TB location. If TB misses are handled in PALcode, there could be an endless loop unless the instruction is held in an instruction buffer or a translated physical PC is maintained by the hardware.

11.10 Address Space Numbers

The Alpha architecture allows a processor to optionally implement address space numbers (process tags) to reduce the need for invalidation of cached address translations for process specific addresses when a context switch occurs. The supported ASN range is 0...MAX_ASN. MAX_ASN is provided in the HWRPB MAX_ASN field. See Section 26.1 for a detailed description of the HWRPB.

Note:

If an ASN outside of the range 0...MAX_ASN is assigned to a process, the operation of the processor is UNDEFINED.

The address space number for the current process is loaded by software in the Address Space Number (ASN) internal processor register with a Swap Privileged Context instruction. ASNs are processor specific and the hardware makes no attempt to maintain coherency across multiple processors. In a multiprocessor system, software is responsible for ensuring the consistency of TB entries for processes that might be rescheduled on different processors.

Systems that support ASNs should have MAX_ASN in the range 13...65535. The number of ASNs should be determined by the market a system is targeting.

Programming Note:

System software should not assume that the number of ASNs is a power of two. This allows, for example, hardware to use N TB tag bits to encode $(2^{**N})-3$ ASN values, one value for ASM=1 PTEs, and one for invalid.

There are several possible ways of using ASNs that result from several complications in a multiprocessor system. Consider the case in which a process that executed on processor 1 is rescheduled on processor 2. If a page is deleted or its protection is changed, the TB in processor 1 has stale data. One solution is to send an interprocessor interrupt to all the processors on which this process could have run and cause them to invalidate the changed PTE. That results in significant overhead in a system with several processors. Another solution is to have software invalidate all TB entries for a process on a new processor before it can begin execution, if the process executed on another processor during its previous execution. That ensures the deletion of possibly stale TB entries on the new processor. A third solution is to assign a new ASN whenever a process is run on a processor that is not the same as the last processor on which it ran.

11.11 Memory Management Faults

Five types of faults are associated with memory access and protection:

- Access Control Violation (ACV)

Taken when the protection field of the Level3 PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode. An Access Control Violation fault is also taken if the KRE bit is zero in an invalid Level1, or Level2 PTE.

For reduced page table regions, ACV taken when the protection field of the Level2 PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode. An Access Control Violation fault is also taken if the KRE bit is zero in an invalid Level1 PTE.

- Fault on Read (FOR)

Occurs when a read is attempted with PTE<FOR> set.

- Fault on Write (FOW)

Occurs when a write is attempted with PTE<FOW> set.

- Fault on Execute (FOE)

Occurs when instruction execution is attempted with PTE<FOE> set.

- Translation Not Valid (TNV)

Taken when a read or write reference is attempted through an invalid PTE in a Level1, Level2, or Level3 page table.

See Section 14.3.1 for a detailed description of these faults.

Those five faults have distinct vectors in the System Control Block. The Access Violation (ACV) fault takes precedence over the faults TNV, FOR, FOW, and FOE. The Translation Not Valid (TNV) fault takes precedence over the faults FOR, FOW, and FOE.

The faults FOR and FOW can occur simultaneously in the CALL_PAL queue instructions, in which case the order that the exceptions are taken is UNPREDICTABLE (see Section 10.1).

Chapter 12

Process Structure (II-A)

12.1 Process Definition

A process is the basic entity that is scheduled for execution by the processor. A process represents a single thread of execution and consists of an address space and both hardware and software context.

The hardware context of a process is defined by:

- Thirty-one integer registers and 31 floating-point registers
- Processor Status (PS)
- Program Counter (PC)
- Four stack pointers
- Asynchronous System Trap Enable and summary registers (ASTEN, ASTSR)
- Process Page Table Base Register (PTBR)
- Address Space Number (ASN)
- Floating Enable Register (FEN)
- Charged Process Cycles
- Process Unique value
- Data Alignment Trap (DAT)
- Performance Monitoring Enable Register (PME)

The software context of a process is defined by operating system software and is system dependent.

A process may share the same address space with other processes or have an address space of its own. There is, however, no separate address space for system software, and therefore, the operating system must be mapped into the address space of each process (see Chapter 11).

In order for a process to execute, its hardware context must be loaded into the integer registers, floating-point registers, and internal processor registers. When a process is being executed, its hardware context is continuously updated. When a process is not being executed, its hardware context is stored in memory.

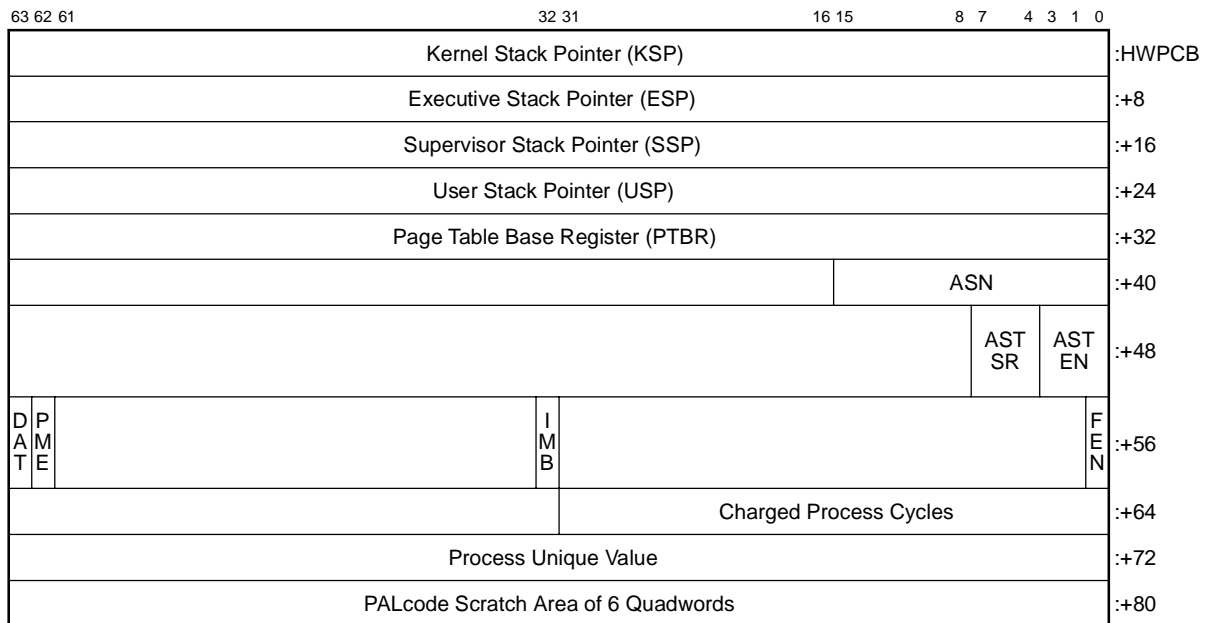
Saving the hardware context of the current process in memory, followed by loading the hardware context for a new process, is termed context switching. Context switching occurs as one process after another is scheduled by the operating system for execution.

12.2 Hardware Privileged Process Context

The hardware context of a process is defined by a privileged part that is context switched with the Swap Privileged Context instruction (SWPCTX) (see Section 10.6), and a nonprivileged part that is context switched by operating system software.

When a process is not executing, its privileged context is stored in a 128-byte naturally aligned memory structure called the Hardware Privileged Context Block (HWPCB). (See Figure 12–1.)

Figure 12–1 Hardware Privileged Context Block



The Hardware Privileged Context Block (HWPCB) for the current process is specified by the Privileged Context Block Base register (PCBB). (See Section 13.3.11.)

The Swap Privileged Context instruction (SWPCTX) saves the privileged context of the current process into the HWPCB specified by PCBB, loads a new value into PCBB, and then loads the privileged context of the new process into the appropriate hardware registers.

The new value loaded into PCBB, as well as the contents of the Privileged Context Block, must satisfy certain constraints or an UNDEFINED operation results:

- The physical address loaded into PCBB must be 128-byte aligned and describes 16 contiguous quadwords that are in a memory-like region. (See Section 5.2.4.)
- The value of PTBR must be the Page Frame Number of an existent page that is in a memory-like region.

It is the responsibility of the operating system to save and load the nonprivileged part of the hardware context.

The SWPCTX instruction returns ownership of the current HWPCB to operating system software and passes ownership of the new HWPCB from the operating system to the processor. Any attempt to write a HWPCB while ownership resides with the processor has UNDEFINED results. If the HWPCB is read while ownership resides with the processor, it is

UNPREDICTABLE whether the original or an updated value of a field is read. The processor can update an HWPCB field at any time. The decision as to whether or not a field is updated is made individually for each field.

If ASNs are not implemented, the ASN field is not read or written by PALcode.

The FEN bit reflects the setting of the FEN IPR.

Setting the PME bit alerts any performance hardware or software in the system to monitor the performance of this process.

The IMB bit records that an IMB was issued in user mode.

The DAT bit controls whether data alignment traps that are fixed up in PALcode are reported to the operating system. If the bit is clear, the trap is reported. If the bit is set, after the fixup, return is to the user. See Section 14.6.

The Charged Process Cycles is the total number of PCC register counts that are charged to the process (modulo $2^{**}32$). When a process context is loaded by the SWPCTX instructions, the contents of the PCC count field (PCC_CNT) are subtracted from the contents of HWPCB[64]<31:0> and the result is written to the PCC offset field (PCC_OFF):

$$\text{PCC}\langle 63:32\rangle \leftarrow (\text{HWPCB}[64]\langle 31:0\rangle - \text{PCC}\langle 31:0\rangle)$$

When a process context is saved by the SWPCTX instruction, the charged process cycles is computed by performing an unsigned add of PCC<63:32> and PCC<31:0>. That value is written to HWPCB[64]<31:0>.

Software Programming Note:

The following example returns in R0 the current PCC register count (modulo $2^{**}32$) for a process. Care is taken not to cause an unwanted sign extension.

```
RPCC    R0                ; Read the processor 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 Process Unique value is that value used in support of multithread implementations. The value is stored in the HWPCB when the process is not active. When the process is active, the value may be cached in hardware internal storage or kept only in the HWPCB.

12.3 Asynchronous System Traps (AST)

Asynchronous System Traps (ASTs) are a means of notifying a process of events that are not synchronized with its execution but that must be dealt with in the context of the process with minimum delay.

Asynchronous System Traps (ASTs) interrupt process execution and are controlled by the AST Enable (ASTEN) and AST Summary (ASTSR) internal processor registers. (See Sections 13.3.2 and 13.3.3, respectively.)

The AST Enable register (ASTEN) contains an enable bit for each of the four processor access modes. When the bit corresponding to an access mode is set, ASTs for that mode are enabled. The AST enable bit for an access mode may be changed by executing a Swap AST Enable instruction (SWASTEN; see Section 10.1.13), or by executing a Move to Processor Register instruction specifying ASTEN (MTPR ASTEN; see Section 13.3.2).

The AST Summary Register (ASTSR) contains a pending bit for each of the four processor access modes. When the bit corresponding to an access mode is set, an AST is pending for that mode.

Kernel mode software may request an AST for a particular access mode by executing a Move to Processor Register instruction specifying ASTSR (MTPR ASTSR; see Section 13.3.3).

Hardware or PALcode monitors the state of ASTEN, ASTSR, PS<CM>, and PS<IPL>. If PS<IPL> is less than 2, and there is an AST pending and enabled for an access mode that is less than or equal to PS<CM> (that is, an equal or more privileged access mode), an AST is initiated at IPL 2.

ASTs that are pending and enabled for a less privileged access mode are not allowed to interrupt execution in a more privileged access mode.

12.4 Process Context Switching

Process context switching occurs as one process after another is scheduled for execution by operating system software. Context switching requires the hardware context of one process to be saved in memory followed by the loading of the hardware context for another process into the hardware registers.

The privileged hardware context is swapped with the CALL_PAL Swap Privileged Context instruction (SWPCTX). Other hardware context must be saved and restored by operating system software.

The sequence in which process context is changed is important because the SWPCTX instruction changes the environment in which the context switching software itself is executing. Also, although hardware does not enforce this, it is advisable to execute the actual context switching software in an environment that cannot be context switched (that is, at an IPL high enough that rescheduling cannot occur).

The SWPCTX instruction is the only method provided for loading certain internal processor registers. The SWPCTX instruction always saves the privileged context of the old process and loads the privileged context of a new process. Therefore, a valid HWPCB must be available to save the privileged context of the old process as well as load the privileged context of the new process.

At system initialization, a valid HWPCB is constructed in the Hardware Restart Parameter Block (HWRPB) for the primary processor. (See Section 26.1.) Thereafter, it is the responsibility of operating system software to ensure a valid HWPCB when executing a SWPCTX instruction.

Internal Processor Registers (II–A)

13.1 Internal Processor Registers

This chapter describes the OpenVMS Internal Processor Registers (IPRs). These registers are read and written with Move from Processor Register (MFPR) and Move to Processor Register (MTPR) instructions. See Section 10.6.

Those instructions accept an input operand in R16 and return a result, if any, in R0. Registers R1, R16, and R17 are UNPREDICTABLE after a CALL_PAL MxPR routine. If a CALL_PAL MxPR routine does not return a result in R0, then R0 is also UNPREDICTABLE on return.

Some IPRs (for example, ASTSR, ASTEN, IPL) may be both read and written in a combined operation by performing an MTPR instruction.

Internal Processor Registers may or may not be implemented as actual hardware registers. An implementation may choose any combination of PALcode and hardware to produce the architecturally specified function. Internal Processor Registers are only accessible from kernel mode.

13.2 Stack Pointer Internal Processor Registers

The stack pointers for user, supervisor, and executive stacks are accessible as IPRs through the CALL_PAL MTPR and MFPR instructions. An implementation may retain some or all of these stack pointers only in the HWPCB. In this case, MTPR and MFPR for these registers must access the corresponding PCB locations. However, implementations that have these stack pointers in internal hardware registers are not required to access the corresponding HWPCB locations for MTPR and MFPR. The HWPCB locations get updated when a SWPCTX instruction is executed.

An implementation may also choose to keep the kernel stack pointer (KSP) in an internal hardware register (labeled IPR_KSP); however, this register is not directly accessible through MTPR and MFPR instructions. Because access to the KSP requires kernel mode, the actual KSP is the current mode stack pointer (R30); thus access to KSP is provided through R30, and no MTPR or MFPR access is required. PALcode routines can directly access IPR_KSP as needed.

At system initialization, the value of the KSP is taken from the initial HWPCB (see Section 12.2). Table 13–1 summarizes the IPRs.

13.3 IPR Summary

Table 13–1 Internal Processor Register (IPR) Summary

Register Name	Mnemonic	Access [†]	Input R16	Output R0	Context Switched
Address Space Number	ASN	R	—	Number	Yes
AST Enable	ASTEN	R/W*	Mask	Mask	Yes
AST Summary Register	ASTSR	R/W*	Mask	Mask	Yes
Data Alignment Trap Fixup	DATFX	W	Value	—	Yes
Executive Stack Pointer	ESP	R/W	Address	Address	Yes
Floating-point Enable	FEN	R/W	Value	Value	Yes
Interprocessor Int. Request	IPIR	W	Number	—	No
Interrupt Priority Level	IPL	R/W*	Value	Value	No
Kernel Stack Pointer	KSP	None	—	—	Yes
Machine Check Error Summary	MCES	R/W	Value	Value	No
Performance Monitoring	PERFMON	W*	IMP	IMP	No
Privileged Context Block Base	PCBB	R	—	Address	No
Processor Base Register	PRBR	R/W	Value	Value	No
Page Table Base Register	PTBR	R	—	Frame	Yes
System Control Block Base	SCBB	R/W	Frame	Frame	No
Software Int. Request Register	SIRR	W	Level	—	No
Software Int. Summary Register	SISR	R	—	Mask	No
Supervisor Stack Pointer	SSP	R/W	Address	Address	Yes
System Page Table Base	SYSPTBR	R/W	Value	Value	Yes
TB Check	TBCHK	R	Number	Status	No
TB Invalid. All	TBIA	W	—	—	No
TB Invalid. All Process	TBIAP	W	—	—	No
TB Invalid. Single	TBIS	W	Address	—	No
TB Invalid. Single Data	TBISD	W	Address	—	No
TB Invalid. Single Instruct.	TBISI	W	Address	—	No
User Stack Pointer	USP	R/W	Address	Address	Yes
Virtual Address Boundary	VIRBND	R/W	Address	Address	Yes
Virtual Page Table Base	VPTB	R/W	Address	Address	No
Who-Am-I	WHAMI	R	—	Number	No

[†] Access symbols are defined in Table 13–2.

Table 13–2 Internal Processor Register (IPR) Access Summary

Access Type	Meaning
R	Access by MFPR only.
W	Access by MTPR only.
R/W	Access by MFPR or MTPR.
W*	Read and Write access accomplished by MTPR. See Section 13.1 for details.
R/W*	Access by MFPR or MTPR. Read and Write access accomplished by MTPR. See Section 13.1 for details.
None	Not accessible by MTPR or MFPR; accessed by PALcode routines as needed.

13.3.1 Address Space Number (ASN)

Access:

Read

Operation:

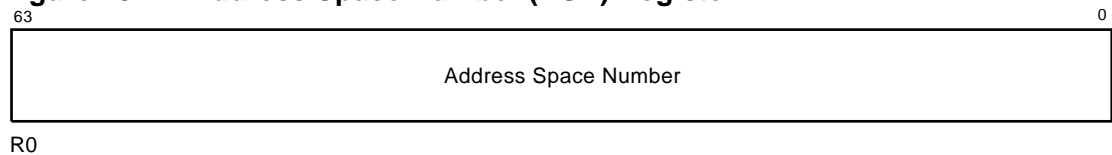
```
IF {ASN are implemented} THEN
  R0 ← ZEXT(ASN)
ELSE
  R0 ← 0
```

Value at System Initialization:

Zero

Format:

Figure 13–1: Address Space Number (ASN) Register



Description:

Address Space Numbers (ASNs) are used to further qualify Translation Buffer references. See Section 11.9. If ASNs are implemented, the current ASN may be read by executing an MFPR instruction specifying ASN.

As processes are scheduled for execution, the ASN for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction. See Section 10.6.7 and Chapter 12.

The ASN register is an implicit operand to the CALL_PAL MFPR_IPR, TBCHK, and TBISx PALcode instructions, in which it is used to qualify the virtual address supplied in R16.

13.3.2 AST Enable (ASTEN)

Access:

Read
Write*

Operation:

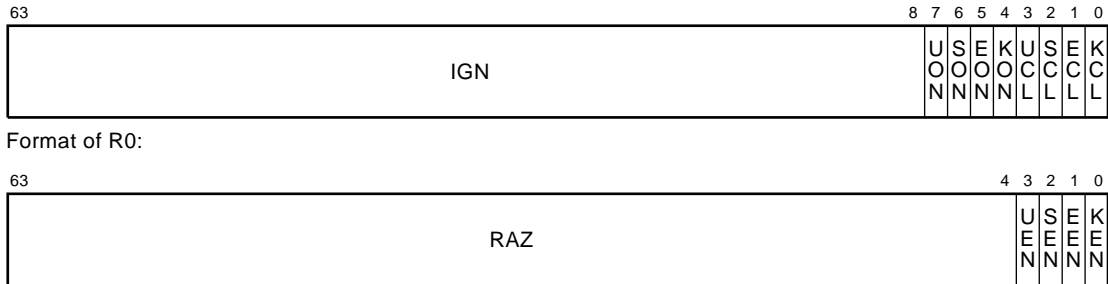
R0 ← ZEXT (ASTEN<3:0>) ! Read (MFPR)
 R0 ← ZEXT (ASTEN<3:0>) ! Write* (MTPR)
 ASTEN<3:0> ← {{ASTEN<3:0> AND R16<3:0>} OR R16<7:4>}
 {check for pending ASTs}

Value at System Initialization:

Zero

Format:

Figure 13–2: AST Enable (ASTEN) Register



Description:

The AST Enable Register records the AST enable state for each of the modes: kernel (KEN), executive (EEN), supervisor (SEN), and user (UEN). By writing R16 appropriately and then executing an MTPR instruction specifying ASTEN, the value of ASTEN may be simultaneously read and modified. R16 contains bit masks that are used to determine the new value of ASTEN:

- Bits R16<0> and R16<4> control the new state of kernel enable.
- Bits R16<1> and R16<5> control the new state of executive enable.
- Bits R16<2> and R16<6> control the new state of supervisor enable.
- Bits R16<3> and R16<7> control the new state of user enable.

An MFPR to ASTEN reads the current value of the ASTEN and returns this value in R0.

An MTPR to ASTEN begins by reading the current value of ASTEN and returning this value in R0. The current value of ASTEN is then ANDed with bits R16<3:0>; these bits preserve (if set to 1) or clear (if equal to 0) the current state of their corresponding enable modes. The value

produced by this operation is then ORed with bits R16<7:4>; these bits turn on (if set to 1) or do not affect (if equal to 0) their corresponding enable modes. The resulting value is then written to the ASTEN.

Note:

All AST enables can be cleared by loading a zero into R16 and executing an MTPR instruction specifying ASTEN. To enable an AST for a given mode, load R16 with a mask that has bits <3:0> set and one of the bits <7:4> corresponding to the AST mode to be set. Then execute an MTPR instruction specifying ASTEN.

As processes are scheduled for execution, the state of the AST enables for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction. The Swap AST Enable (SWASTEN) instruction can be used to change the enable state for the current access mode. See Section 10.1.13 and Chapter 12.

13.3.3 AST Summary Register (ASTSR)

Access:

Read
Write*

Operation:

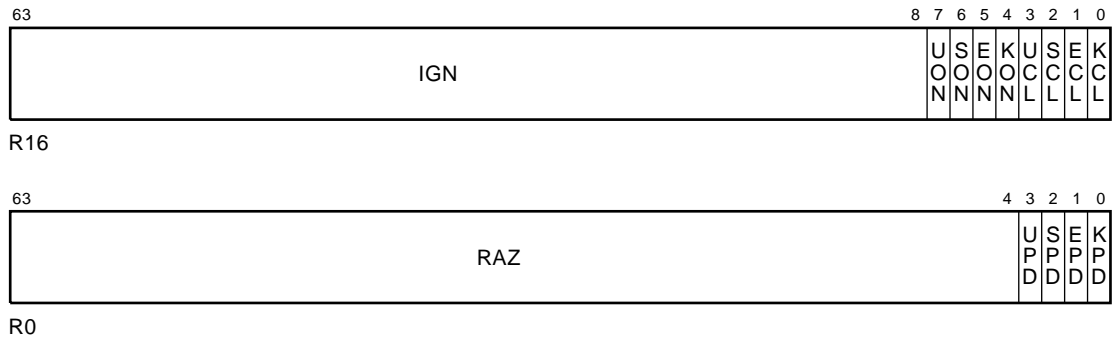
$R0 \leftarrow ZEXT(ASTSR<3:0>)$! Read (MFPR)
 $R0 \leftarrow ZEXT(ASTSR<3:0>)$! Write* (MTPR)
 $ASTSR<3:0> \leftarrow \{ \{ASTSR<3:0> \text{ AND } R16<3:0> \} \text{ OR } R16<7:4> \}$
 {check for pending ASTs}

Value at System Initialization:

Zero

Format:

Figure 13–3: AST Summary Register (ASTSR)



Description:

The AST Summary Register records the AST pending state for each of the modes: kernel (KPD), executive (EPD), supervisor (SPD), and user (UPD).

By writing R16 appropriately and then executing an MTPR instruction specifying ASTSR, the value of ASTSR may be simultaneously read and modified. R16 contains bit masks used to determine the new value of ASTSR:

- Bits R16<0> and R16<4> control the new state of kernel pending.
- Bits R16<1> and R16<5> control the new state of executive pending.
- Bits R16<2> and R16<6> control the new state of supervisor pending.
- Bits R16<3> and R16<7> control the new state of user pending.

An MFPR reads the current value of ASTSR and returns this value in R0.

An MTPR to ASTSR begins by reading the current value of ASTSR and returning this value in R0. The current value of ASTSR is then ANDed with bits R16<3:0>; these bits preserve (if set to 1) or clear (if equal to 0) the current state of their corresponding pending modes. The value

produced by this operation is then ORed with bits R16<7:4>; these bits turn on (if set to 1) or do not affect (if equal to 0) their corresponding pending modes. The resulting value is then written to the ASTSR.

Note:

All AST requests can be cleared by loading a zero in R16 and executing an MTPR instruction specifying ASTSR. To request an AST for a given mode, load R16 with a mask that has bits <3:0> set and one of the bits <7:4> corresponding to the AST mode to be set. Then execute an MTPR instruction specifying ASTSR.

As processes are scheduled for execution, the pending AST state for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction. See Section 10.6.7 and Chapter 12.

When the processor IPL is less than 2, and proper enabling conditions are present, an AST interrupt is initiated at IPL 2 and the corresponding access mode bit in ASTSR is cleared. See Section 14.7.6.

13.3.4 Data Alignment Trap Fixup (DATFX)

Access:

Write

Operation:

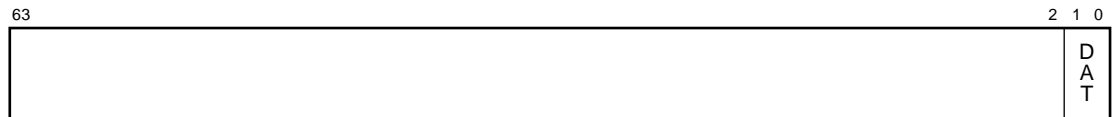
$\text{DATFX} \leftarrow \text{R16} \langle 0 \rangle$
 $(\text{HWPCB} + 56) \langle 63 \rangle \leftarrow \text{DATFX}$

Value at System Initialization:

Zero

Format:

Figure 13–4: Data Alignment Trap Fixup (DATFX)



Description:

Data Alignment traps are fixed up in PALcode and are reported to the operating system under the control of the DAT bit. If the bit is zero, the trap is reported. For the LDx_L and STx_C instructions, no fixup is possible and an illegal operand exception is generated.

For the description of the data alignment traps, see Section 14.6.

13.3.5 Executive Stack Pointer (ESP)

Access:

Read/Write

Operation:

```
IF {internal registers for stack pointers} THEN      ! Read
  R0 ← ESP
ELSE
  R0 ← (IPR_PCBB + HWPCB_ESP)

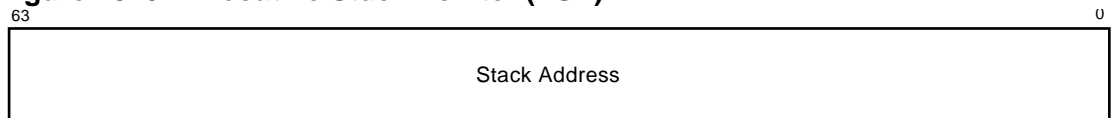
IF {internal registers for stack pointers} THEN      ! Write
  ESP ← R16
ELSE
  (IPR_PCBB + HWPCB_ESP) ← R16
```

Value at System Initialization:

Value in the initial HWPCB

Format:

Figure 13–5: Executive Stack Pointer (ESP)



Description:

This register allows the stack pointer for executive mode (ESP) to be read and written via MFPR and MTPR instructions that specify ESP.

The current stack pointer may be read and written directly by specifying scalar register SP (R30).

As processes are scheduled for execution, the stack pointers for the next process to execute are loaded using the Swap Privileged Context (SWPCTX) instruction. See Section 10.6.7 and Chapter 12.

13.3.6 Floating Enable (FEN)

Access:

Read/Write

Operation:

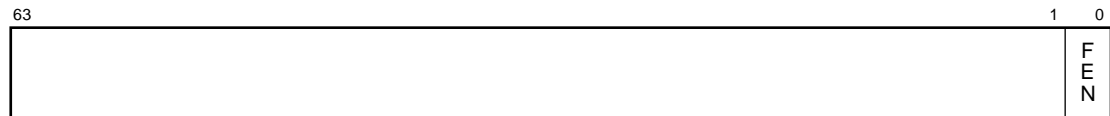
$R0 \leftarrow ZEXT(FEN)$! Read
 $FEN \leftarrow R16<0>$! Write
 $(HWPCB+56)<0> \leftarrow FEN$! Update PCB on Write

Value at System Initialization:

Zero

Format:

Figure 13–6: Floating Enable (FEN) Register



Description:

The floating-point unit can be disabled with the CALL_PAL CLRFEIN instruction. If the Floating Enable Register (FEN) is zero, all instructions that have floating registers as operands cause a floating-point disabled fault. See Section 14.3.1.1.

13.3.7 Interprocessor Interrupt Request (IPIR)

Access:

Write

Operation:

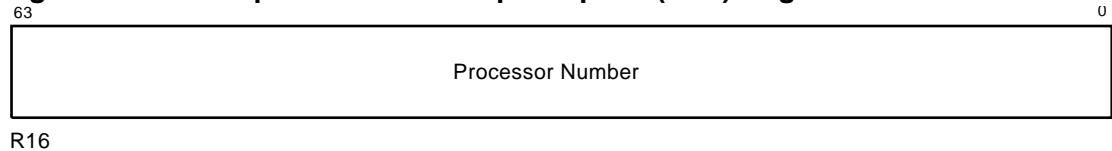
$IPIR \leftarrow R16$

Value at System Initialization:

Not applicable

Format:

Figure 13–7: Interprocessor Interrupt Request (IPIR) Register



Description:

An interprocessor interrupt can be requested on a specified processor by writing that processor's number into the IPIR register through an MTPR instruction. The interrupt request is recorded on the target processor and is initiated when proper enabling conditions are present.

Programming Note:

The interrupt need not be initiated before the next instruction is executed on the requesting processor, even if the requesting processor is also the target processor for the request.

For additional information on interprocessor interrupts, see Section 14.4.6.

13.3.8 Interrupt Priority Level (IPL)

Access:

Read/Write*

Operation:

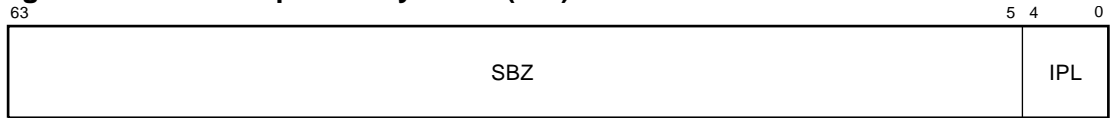
```
R0 ← ZEXT(PS<IPL>)           ! Read
R0 ← ZEXT(PS<IPL>)           ! Write*
PS<IPL> ← R16<4:0>           ! Write
{check for pending ASTs or interrupts}
```

Value at System Initialization:

31

Format:

Figure 13–8: Interrupt Priority Level (IPL)



Description:

An MFPR IPL returns the current interrupt priority level in R0. An MTPR IPL returns the current interrupt priority level in R0 and sets the interrupt priority level to the value in R16. If proper enabling conditions are present, an interrupt or AST is initiated prior to issuing the next instruction. See Sections 14.4.2 and 14.7.6. R16<63:5> are defined as RAZ/SBZ. Therefore, the presence of nonzero bits upon write in R16<63:5> may cause UNDEFINED results.

13.3.9 Machine Check Error Summary Register (MCES)

Access:

Read/Write

Operation:

```

R0 ← ZEXT(MCES)                                ! Read

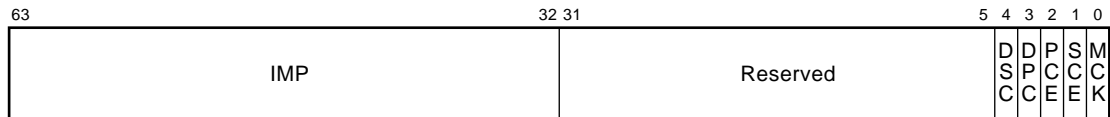
IF {R16<0> EQ 1} THEN MCES<0> ← 0             ! Write
IF {R16<1> EQ 1} THEN MCES<1> ← 0
IF {R16<2> EQ 1} THEN MCES<2> ← 0
MCES<3> ← R16<3>
MCES<4> ← R16<4>
    
```

Value at System Initialization:

Zero

Format:

Figure 13–9: Machine Check Error Summary (MCES) Register



Description:

The use of the MCES IPR is described in Section 14.5.

MCK (MCES<0>) is set by the hardware or PALcode when a processor or system machine check occurs. SCE (MCES<1>) is set by the hardware or PALcode when a system correctable error occurs. PCE (MCES<2>) is set by the hardware or PALcode when a processor correctable error occurs.

Setting the corresponding bit(s) in R16 clears MCK, SCE, and PCE. MCK is cleared by the operating system machine check error handler and used by the hardware or PALcode to detect double machine checks. SCE and PCE are cleared by the operating system or processor system correctable error handlers; these bits are used to indicate that the associated correctable error logout area may be reused by hardware or PALcode. In the event of double correctable errors, PALcode does not overwrite the logout area and does not force the processor to enter console I/O mode. See Section 14.5.1.

DPC (MCES<3>) and DSC (MCES<4>) are used to disable reporting of correctable errors to system software. The generation and correction of the machine check are not affected; only the report to system software is disabled. Setting DPC disables reporting of processor-correctable machine checks. Setting DSC disables reporting of system-correctable machine checks. Implementation-dependent (IMP) bits may be used to report implementation-specific errors.

13.3.10 Performance Monitoring Register (PERFMON)

Access:

Write*

Operation:

- ! R16 contains implementation specific input values
- ! R17 contains implementation specific input values
- ! R0 may return implementation specific values
- ! Operations and actions taken are implementation specific

Value at System Initialization:

Implementation Dependent

Format:

Figure 13–10: Performance Monitoring (PERFMON) Register



Description:

The arguments and actions of this performance monitoring function are platform and chip dependent. The functions, when defined for an implementation, are described in Appendix E.

R16 and R17 contain implementation-dependent input values. Implementation-specific values may be returned in R0.

13.3.11 Privileged Context Block Base (PCBB)

Access:

Read

Operation:

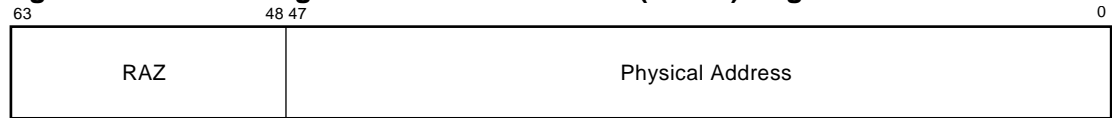
$R0 \leftarrow ZEXT(PCBB)$

Value at System Initialization:

Address of processor's bootstrap HWPCB

Format:

Figure 13–11: Privileged Context Block Base (PCBB) Register



R0

Description:

The Privileged Context Block Base Register contains the physical address of the privileged context block for the current process. It may be read by executing an MFPR instruction specifying PCBB.

PCBB is written by the Swap Privileged Context (SWPCTX) instruction. See Section 10.6.7 and Chapter 12.

13.3.12 Processor Base Register (PRBR)

Access:

Read/Write

Operation:

$R0 \leftarrow PRBR$! Read

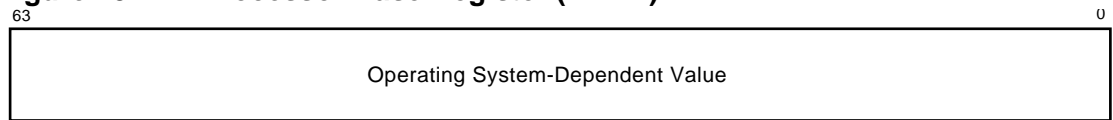
$PRBR \leftarrow R16$! Write

Value at System Initialization:

UNPREDICTABLE

Format:

Figure 13–12: Processor Base Register (PRBR)



Description:

In a multiprocessor system, it is desirable for the operating system to be able to locate a processor-specific data structure in a simple and straightforward manner. The Processor Base Register provides a quadword of operating system-dependent state that can be read and written via MFPR and MTPR instructions that specify PRBR.

13.3.13 Page Table Base Register (PTBR)

Access:

Read

Operation:

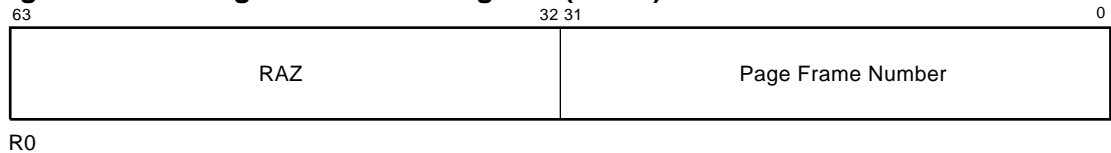
$R0 \leftarrow PTBR$

Value at System Initialization:

Value in the bootstrap HWPCB

Format:

Figure 13–13: Page Table Base Register (PTBR)



Description:

The Page Table Base Register contains the page frame number of the first-level page table for the current process. It may be read by executing an MFPR instruction specifying PTBR. See Chapter 11.

As processes are scheduled for execution, the PTBR for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction. See Section 10.6.7 and Chapter 12.

13.3.14 System Control Block Base (SCBB)

Access:

Read/Write

Operation:

$R0 \leftarrow ZEXT(SCBB)$! Read

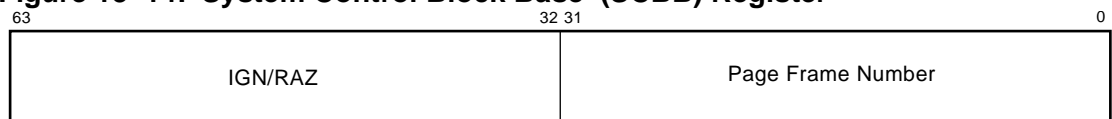
$SCBB \leftarrow R16$! Write

Value at System Initialization:

UNPREDICTABLE

Format:

Figure 13–14: System Control Block Base (SCBB) Register



Description:

The System Control Block Base Register holds the Page Frame Number (PFN) of the System Control Block, which is used to dispatch exceptions and interrupts, and may be read and written by executing MFPR and MTPR instructions that specify SCBB. See Section 14.6.

When SCBB is written, the specified physical address must be the PFN of a page that is neither in I/O space nor nonexistent memory, or UNDEFINED operation will result.

13.3.15 Software Interrupt Request Register (SIRR)

Access:

Write

Operation:

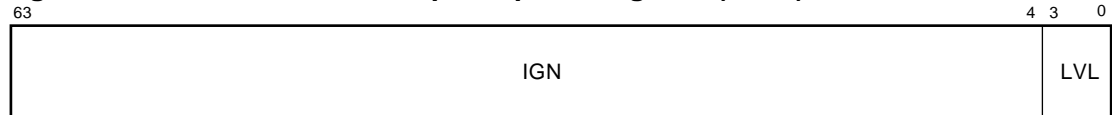
```
IF R16<3:0> NE 0 THEN
  SISR<R16<3:0>> ← 1
```

Value at System Initialization:

Not applicable

Format:

Figure 13–15: Software Interrupt Request Register (SIRR)



R16

Description:

A software interrupt may be requested for a particular Interrupt Priority Level (IPL) by executing an MTPR instruction specifying SIRR. Software interrupts may be requested at levels 0 through 15 (requests at level 0 are ignored).

An MTPR SIRR sets the bit corresponding to the specified interrupt level in the Software Interrupt Summary Register (SISR).

If proper enabling conditions are present, a software interrupt is initiated prior to issuing the next instruction. See Sections 14.4.1 and 14.7.6.

13.3.17 Supervisor Stack Pointer (SSP)

Access:

Read/Write

Operation:

```
IF {internal registers for stack pointers} THEN      ! Read
  R0 ← SSP
ELSE
  R0 ← (IPR_PCBB + HWPCB_SSP)

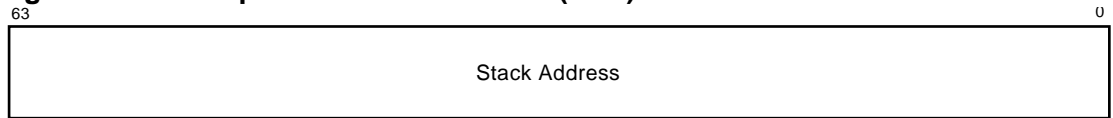
IF {internal registers for stack pointers} THEN      ! Write
  SSP ← R16
ELSE
  (IPR_PCBB + HWPCB_SSP) ← R16
```

Value at System Initialization:

Value in the initial HWPCB

Format:

Figure 13–17: Supervisor Stack Pointer (SSP)



Description:

The Supervisor Stack Pointer register allows the stack pointer for supervisor mode (SSP) to be read and written by using MFPR and MTPR instructions that specify SSP.

The current stack pointer may be read and written directly by specifying scalar register SP (R30).

As processes are scheduled for execution, the stack pointers for the next process to execute are loaded using the Swap Privileged Context (SWPCTX) instruction. See Section 10.6.7 and Chapter 12.

13.3.19 Translation Buffer Check (TBCHK)

Access:

Read

Operation:

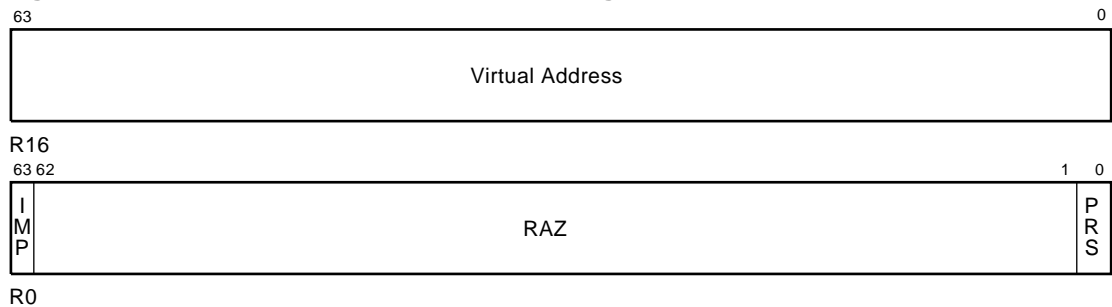
```
R0 ← 0
IF {implemented} THEN
    R0<0> ← {indicator that VA in R16 is in TB}
ELSE
    R0<63> ← 1
```

Value at System Initialization:

Correct results are always returned

Format:

Figure 13–19: Translation Buffer Check Register (TBCHK)



Description:

The Translation Buffer Check Register provides the capability to determine if a virtual address is present in the Translation Buffer by executing an MFPR instruction specifying TBCHK. See Section 11.9.

The virtual address to be checked is specified in R16 and may be any address within the desired page. If ASNs are implemented, only those Translation Buffer entries that are associated with the current value of the ASN IPR will be checked for the virtual address. The value read contains an indication of whether the function is implemented and whether the virtual address is present in the Translation Buffer.

If the function is not implemented, a one is returned in bit <63> and bit <0> is clear. Otherwise, bit <63> is clear and bit <0> indicates the presence or absence of the virtual address in the Translation Buffer. Bit <0> set indicates the virtual address is present; bit <0> clear indicates it is absent.

The TBCHK register can be used by system software for working set management.

13.3.20 Translation Buffer Invalidate All (TBIA)

Access:

Write

Operation:

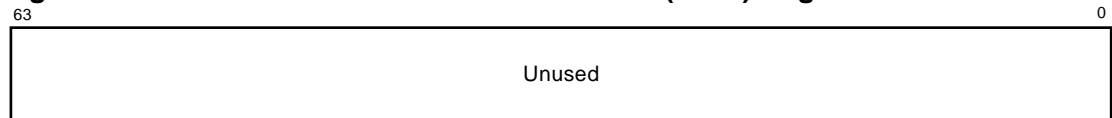
{Invalidate all TB entries}

Value at System Initialization:

Not applicable

Format:

Figure 13–20: Translation Buffer Invalidate All (TBIA) Register



R16

Description:

The Translation Buffer Invalidate All Register provides the capability to invalidate all entries in the Translation Buffer by executing an MTPR instruction specifying TBIA. See Section 11.9 for information on translation buffers.

13.3.21 Translation Buffer Invalidate All Process (TBIAP)

Access:

Write

Operation:

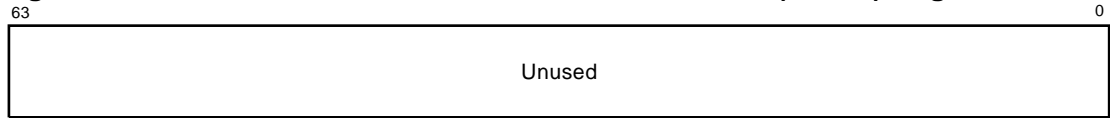
{Invalidate all TB entries with PTE<ASM> clear}

Value at System Initialization:

Not applicable

Format:

Figure 13–21: Translation Buffer Invalidate All Process (TBIAP) Register



R16

Description:

The Translation Buffer Invalidate All Process Register provides the capability to invalidate all entries in the Translation Buffer that do not have the ASM bit set by executing an MTPR instruction specifying TBIAP. See Section 11.9 for information on translation buffers and Section 11.10 for information on address space numbers (ASNs), because ASNs can implicitly modify TB operations.

Notes:

More entries may be invalidated by this operation. For example, some implementations may flush the entire TB on a TBIAP.

13.3.22 Translation Buffer Invalidate Single (TBISx)

Access:

Write

Operation:

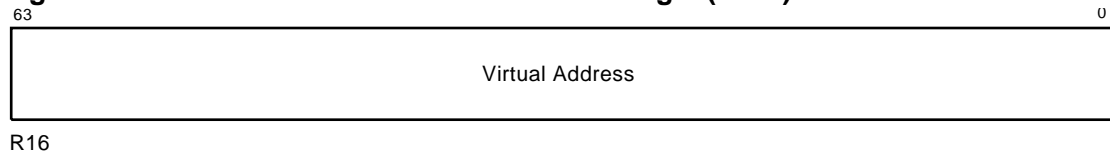
TBIS:
 {Invalidate single Data TB entry using R16}
 {Invalidate single Instruction TB entry using R16}
TBISD:
 {Invalidate single Data TB entry using R16}
TBISI:
 {Invalidate single Instruction TB entry using R16}

Value at System Initialization:

Not applicable

Format:

Figure 13–22: Translation Buffer Invalidate Single (TBIS)



Description:

The Translation Buffer Invalidate Single Registers provide the capability to invalidate a single entry in the Instruction Translation Buffer (TBISI), the Data Translation Buffer (TBISD), or both translation buffers (TBIS). The virtual address to be invalidated is passed in R16 and may be any address within the desired page. See Section 11.9 for information on translation buffers and Section 11.10 for information on address space numbers (ASNs), because ASNs can implicitly modify TB operations.

Notes:

- More than the single entry may be invalidated by this operation. For example some implementations may flush the entire TB on a TBIS. As a result, if the specified address does not match any entry in the Translation Buffer, then it is implementation dependent whether the state of the Translation Buffer is affected by the operation.

13.3.23 User Stack Pointer (USP)

Access:

Read/Write

Operation:

```
IF {internal registers for stack pointers} THEN      ! Read
  R0 ← USP
ELSE
  R0 ← (IPR_PCBB + HWPCB_USP)

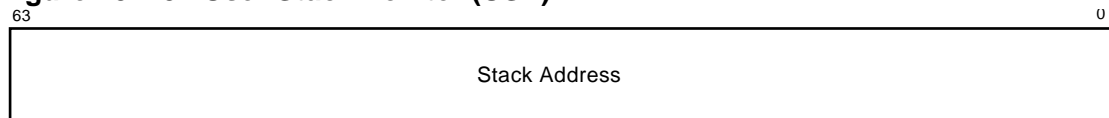
IF {internal registers for stack pointers} THEN      ! Write
  USP ← R16
ELSE
  (IPR_PCBB + HWPCB_USP) ← R16
```

Value at System Initialization:

Value in the initial HWPCB

Format:

Figure 13–23: User Stack Pointer (USP)



Description:

This register allows the stack pointer for user mode (USP) to be read and written via MFPR and MTPR instructions that specify USP.

The current stack pointer may be read and written directly by specifying scalar register SP (R30).

As processes are scheduled for execution, the stack pointers for the next process to execute are loaded using the Swap Privileged Context (SWPCTX) instruction. See Section 10.6.7 and Chapter 12.

13.3.24 Virtual Address Boundary Register (VIRBND)

Access:

Read/Write

Operation:

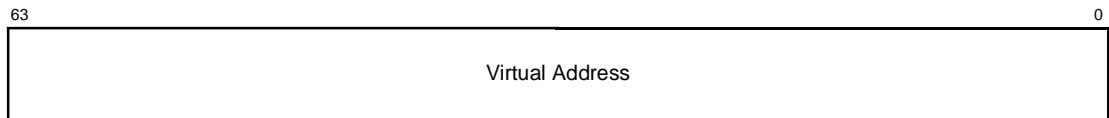
```
R0 <- VIRBND          ! Read
VIRBND <- R.16        ! Write
```

Value at System Initialization:

-1

Format:

Figure 13–24: Virtual Address Boundary (VIRBND) Register



Description:

The Virtual Address Boundary Register holds the address used to determine which page table physical base register is used during address translation, either PTBR or SYSPTBR. It may be read and written by executing MFPR and MTPR instructions that specify VIRBND.

UNPREDICTABLE operations result if the address is not 64-bit aligned. At Processor Initialization, VIRBND is initialized to a value of -1, thereby forcing all translations to use PTBR. The value in SYSPTBR is effectively ignored. Section 11.8 further describes the use of this register.

Implementation of VIRBND and SYSPTBR is optional. If not implemented, only PTBR is used as a base during address translation.

13.3.25 Virtual Page Table Base (VPTB)

Access:

Read/Write

Operation:

R0 ← VPTB ! Read

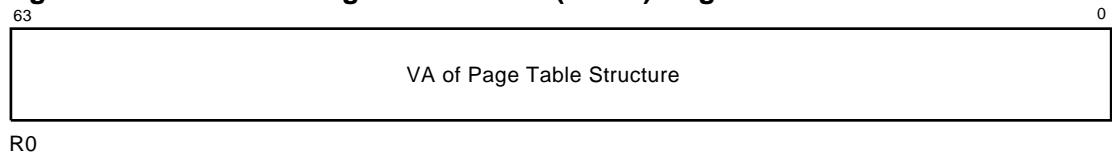
VPTB ← R16 ! Write

Value at System Initialization:

Initialized by the console in the bootstrap address space.

Format:

Figure 13–25: Virtual Page Table Base (VPTB) Register



Description:

The Virtual Page Table Base Register contains the virtual address of the base of the entire three-level page table structure. It may be read by executing an MFPR instruction specifying VPTB. It is written at system initialization using an MTPR instruction specifying VPTB. See Section 11.8.2 and Section 27.4 for initialization considerations.

13.3.26 Who-Am-I (WHAMI)

Access:

Read

Operation:

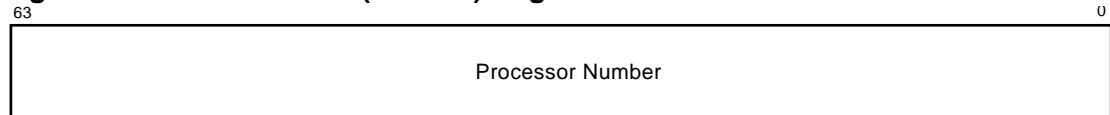
$R0 \leftarrow WHAMI$

Value at System Initialization:

Processor number

Format:

Figure 13–26: Who-Am-I (WHAMI) Register



R0

Description:

The Who-Am-I Register provides the capability to read the current processor number by executing an MFPR instruction specifying WHAMI. The processor number returned is in the range 0 to the number of processors minus one that can be configured in the system. Processor number FFFF FFFF FFFF FFFF₁₆ is reserved.

The current processor number is useful in a multiprocessing system to index arrays that store per processor information. Such information is operating system dependent.

Exceptions, Interrupts, and Machine Checks (II–A)

14.1 Introduction

At certain times during the operation of a system, events within the system require the execution of software outside the explicit flow of control. When such an exceptional event occurs, an Alpha processor forces a change in control flow from that indicated by the current instruction stream. The notification process for such events is of one of three types:

- Exceptions

These events are relevant primarily to the currently executing process and normally invoke software in the context of the current process. The three types of exceptions are faults, arithmetic traps, and synchronous traps. Exceptions are described in Section 14.3.

- Interrupts

These events are primarily relevant to other processes or to the system as a whole and are typically serviced in a system-wide context.

Some interrupts are of such urgency that they require high-priority service, while others must be synchronized with independent events. To meet these needs, each processor has priority logic that grants interrupt service to the highest priority event at any point in time. Interrupts are described in Section 14.4.

- Machine Checks

These events are generally the result of serious hardware failure. The registers and memory are potentially in an indeterminate state such that the instruction execution cannot necessarily be correctly restarted, completed, simulated, or undone. Machine checks are described in Section 14.5.

For all such events, the change in flow of control involves changing the Program Counter (PC), possibly changing the execution mode (current mode) and/or interrupt priority level (IPL) in the Processor Status (PS), and saving the old values of the PC and PS. The old values are saved on the target stack as part of an Exception, Interrupt, or Machine Check Stack Frame. Collectively, those elements are described in Section 14.2.

The service routines that handle exceptions, interrupts, and machine checks are specified by entry points in the System Control Block (SCB), described in Section 14.6.

Return from an exception, interrupt, or machine check is done via the CALL_PAL REI instruction. As part of its work, CALL_PAL REI restores the saved values of PC and PS and pops them off the stack.

14.1.1 Differences Between Exceptions, Interrupts, and Machine Checks

Generally, exceptions, interrupts, and machine checks are similar. However, there are four important differences:

1. An exception is caused by the execution of an instruction. An interrupt is caused by some activity in the system that may be independent of any instruction. A machine check is associated with a hardware error condition.
2. The IPL of the processor is not changed when the processor initiates an exception. The IPL is always raised when an interrupt is initiated. The IPL is always raised when a machine check is initiated, and for all machine checks other than system correctable, is raised to 31 (highest priority level). (For system correctable machine checks, the IPL is raised to 20.)
3. Exceptions are always initiated immediately, no matter what the processor IPL is. Interrupts are deferred until the processor IPL drops below the IPL of the requesting source. Machine checks can be initiated immediately or deferred, depending on error conditions.
4. Some exceptions can be selectively disabled by selecting instructions that do not check for exception conditions. If an exception condition occurs in such an instruction, the condition is totally ignored and no state is saved to signal that condition at a later time.

If an interrupt request occurs while the processor IPL is equal to or greater than that of the interrupting source, the condition will eventually initiate an interrupt if the interrupt request is still present and the processor IPL is lowered below that of the interrupting source.

Machine checks cannot be disabled. Machine checks can be initiated immediately or deferred, depending on the error condition. Also, they can be deliberately generated by software.

14.1.2 Exceptions, Interrupts, and Machine Checks Summary

Table 14–1 summarizes the actions taken on an exception, interrupt, or machine check. The remaining sections in this chapter describe those actions in greater detail.

- The "SavedPC" column describes what is saved in the "PC" field of the exception or interrupt or machine check stack frame.
 1. "Current" indicates the PC of the instruction at which the exception or interrupt or machine check was taken,
 2. "Next" indicates the PC of the successor instruction.
- The "NewMode" column specifies the mode and stack that the exception or interrupt or machine check routine will start with. For change mode traps, "MostPrv" indicates the more privileged of the current and new modes.
- The "R2" column specifies the value with which R2 is loaded, after its original value has been saved in the exception or interrupt or machine check stack frame. The SCB vector quadword, "SCBv", is loaded into R2 for all interrupts and exceptions and machine checks.

- The "R3" column specifies the value with which R3 is loaded, after its original value has been saved in the exception or interrupt or machine check stack frame. The SCB parameter quadword, "SCBp", is loaded into R3 for all interrupts and exceptions and machine checks.
- The "R4" column specifies the value with which R4 is loaded, after its original value has been saved in the exception or interrupt or machine check stack frame. If the "R4" column is blank, the value in R4 is UNPREDICTABLE on entry to an interrupt or exception.
 1. "VA" indicates the exact virtual address that triggered a memory management fault or data alignment trap.
 2. "Mask" indicates the Register Write Mask.
 3. "LAOff" indicates the offset from the base of the logout area in the HWRPB (see Section 14.5.2).
- The "R5" column specifies the value with which R5 is loaded, after its original value has been saved in the exception or interrupt or machine check stack frame. If the "R5" column is blank, the value in R5 is UNPREDICTABLE on entry to an interrupt or exception or machine check.
 1. "MMF" indicates the Memory Management Flags.
 2. "Exc" indicates the Exception Summary parameter.
 3. "RW" indicates Read/Load =0 Write/Store =1 for data alignment traps

Table 14–1 Exceptions, Interrupts, and Machine Checks Summary

	SavedPC	NewMode	R2	R3	R4	R5
Exceptions – Faults :						
Floating Disabled Fault	Current	Kernel	SCBv	SCBp		
Memory Management Faults :						
Access Control Violation	Current	Kernel	SCBv	SCBp	VA	MMF
Translation Not Valid	Current	Kernel	SCBv	SCBp	VA	MMF
Fault on Read	Current	Kernel	SCBv	SCBp	VA	MMF
Fault on Write	Current	Kernel	SCBv	SCBp	VA	MMF
Fault on Execute	Current	Kernel	SCBv	SCBp	VA	MMF
Exceptions – Arithmetic Traps:						
Arithmetic Traps	Next	Kernel	SCBv	SCBp	Mask	Exc
Exceptions - Synchronous Traps :						
Breakpoint Trap	Next	Kernel	SCBv	SCBp		
Bugcheck Trap	Next	Kernel	SCBv	SCBp		
Change Mode to K/E/S/U	Next	MostPrv	SCBv	SCBp		

Table 14–1 Exceptions, Interrupts, and Machine Checks Summary (Continued)

	SavedPC	NewMode	R2	R3	R4	R5
Exceptions - Synchronous Traps, Continued :						
Illegal Instruction	Next	Kernel	SCBv	SCBp		
Illegal Operand	Next	Kernel	SCBv	SCBp		
Data Alignment Trap	Next	Kernel	SCBv	SCBp	VA	RW
Interrupts :						
Asynch System Trap (4)	Current	Kernel	SCBv	SCBp		
Interval Clock	Current	Kernel	SCBv	SCBp		
Interprocessor Interrupt	Current	Kernel	SCBv	SCBp		
Software Interrupts	Current	Kernel	SCBv	SCBp		
Performance monitor	Current	Kernel	SCBv	SCBp	IMP	IMP
Passive Release	Current	Kernel	SCBv	SCBp		
Powerfail	Current	Kernel	SCBv	SCBp		
I/O Device	Current	Kernel	SCBv	SCBp		
Machine Checks :						
Processor Correctable	Current	Kernel	SCBv	SCBp	LAOff	
System Correctable	Current	Kernel	SCBv	SCBp	LAOff	
System	Current	Kernel	SCBv	SCBp	LAOff	
Processor	Current	Kernel	SCBv	SCBp	LAOff	

14.2 Processor State and Exception/Interrupt/Machine Check Stack Frame

Processor state consists of a quadword of privileged information called the Processor Status (PS) and a quadword containing the Program Counter (PC), which is the virtual address of the next instruction.

When an exception, interrupt, or machine check is initiated, the current processor state during the exception, interrupt, or machine check must be preserved. This is accomplished by automatically pushing the PS and the PC on the target stack.

Subsequently, instruction execution can be continued at the point of the exception, interrupt, or machine check by executing a CALL_PAL REI instruction (see Section 10.1.11).

Process context such as memory mapping information is not saved or restored on each exception, interrupt, or machine check. Instead, it is saved and restored when process context switching is performed. Other processor status is changed even less frequently (see Chapter 12).

14.2.1 Processor Status

The PS can be explicitly read with the CALL_PAL RD_PS instruction. The PS<SW> field can be explicitly written with the CALL_PAL WR_PS_SW instruction. See Section 10.1.

The terms current PS and saved PS are used to distinguish between this status information when it is stored internal to the processor and when copies of it are materialized in memory. The current PS is shown in Figure 14–1, the saved PS in Figure 14–2, and the bits for both are described in Table 14–2.

Figure 14–1: Current Processor Status (PS Register)

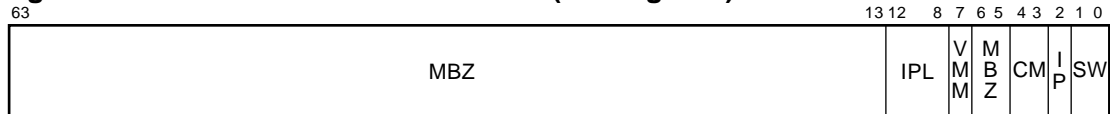


Figure 14–2: Saved Processor Status (PS on Stack)

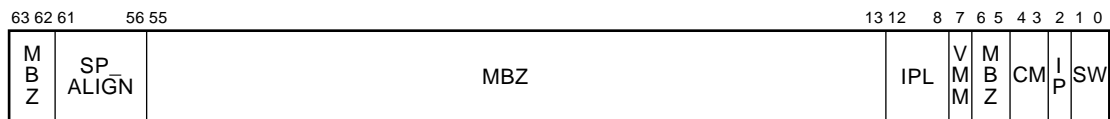


Table 14–2 Processor Status Register Summary

Bits	Description
63–62	Reserved to Compaq, MBZ.
61–56	Stack alignment (SP_ALIGN) The previous stack byte alignment within a 64-byte aligned area, in the range 0 to 63. This field is set in the saved PS during the act of taking an exception or interrupt; it is used by the CALL_PAL REI instruction to restore the previous stack byte alignment.
55–13	Reserved to Compaq, MBZ.
12–8	Interrupt priority level (IPL) The current processor priority, in the range 0 to 31.
7	Virtual machine monitor (VMM). When set, the processor is executing in a virtual machine monitor. When clear, the processor is running in either real or virtual machine mode. Programming Note: This bit is only meaningful when running with PALcode that implements virtual machine capabilities.
6–5	Reserved to Compaq, MBZ.

Table 14–2 Processor Status Register Summary (Continued)

Bits	Description								
4–3	<p>Current mode (CM)</p> <p>The access mode of the currently executing process as follows:</p> <table> <tr> <td>0</td> <td>Kernel</td> </tr> <tr> <td>1</td> <td>Executive</td> </tr> <tr> <td>2</td> <td>Supervisor</td> </tr> <tr> <td>3</td> <td>User</td> </tr> </table>	0	Kernel	1	Executive	2	Supervisor	3	User
0	Kernel								
1	Executive								
2	Supervisor								
3	User								
2	<p>Interrupt pending (IP)</p> <p>Set when an interrupt (software or hardware but not AST) is initiated; indicates an interrupt is in progress.</p>								
1–0	<p>Reserved for Software (SW)</p> <p>These bits are reserved for software use and can be read and written at any time by the software, regardless of the current mode. The value of these bits is ignored by the hardware. The software field is set to zero at the initiation of either an exception or an interrupt.</p>								

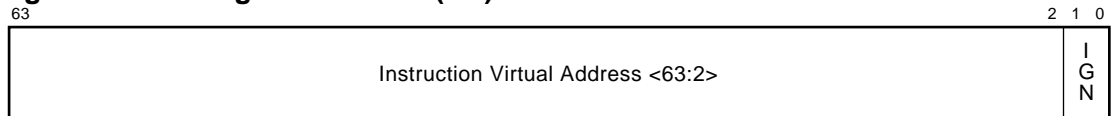
At bootstrap, the initial value of PS is set to $1F00_{16}$. Previous stack alignment is zero, IPL is 31, VMM is clear, CM is kernel, and the SW and IP fields are zero.

14.2.2 Program Counter

The PC (Figure 14–3) is a 64-bit virtual address. All instructions are aligned on longword boundaries and, therefore, hardware can assume zero for the two low-order PC bits. The PC is discussed in Section 14.2.6.

The PC can be explicitly read with the Unconditional Branch (BR) instruction. All branching instructions also load a new value into the PC.

Figure 14–3: Program Counter (PC)



14.2.3 Processor Interrupt Priority Level (IPL)

Each processor has 32 interrupt priority levels (IPLs) divided into 16 software levels (numbered 0 to 15), and 16 hardware levels (numbered 16 to 31). User applications and most operating system software run at IPL 0, which may be thought of as process level. Higher numbered interrupt levels have higher priority; that is, any request at an interrupt level higher than the processor's current IPL will interrupt immediately, but requests at lower or equal levels are deferred.

Interrupt levels 0 to 15 exist solely for use by software. No hardware event can request an interrupt on these levels. Conversely, interrupt levels 16 to 31 exist solely for use by hardware. Serious system failures, such as a machine check abort, however, raise the IPL to the highest level (31) to minimize processor interruption until the problem is corrected, and execute in kernel mode on the kernel stack.

14.2.4 Protection Modes

Each processor has four protection modes: kernel, executive, supervisor, and user. Per-page memory protection varies as a function of mode (for example, a page can be made read-only in user mode, but read-write in supervisor, executive, or kernel mode).

For each process, a separate stack is associated with each mode. Corruption of one stack does not affect use of the other stacks.

Some instructions, termed privileged instructions, may be executed only in kernel mode.

14.2.5 Processor Stacks

Each processor has four stacks. There are four process-specific stacks associated with the four modes of the current process. At any given time, only one of these stacks is actively used as the current stack.

14.2.6 Stack Frames

When an exception, interrupt, or machine check occurs, a stack frame (Figure 14–3) is pushed on the target stack. Regardless of the type of event notification, this stack frame consists of a 64-byte-aligned structure that contains the saved contents of registers R2..R7, the Program Counter (PC), and the Processor Status (PS). Registers R2 and R3 are then loaded with vector and parameter from the SCB for the exception, interrupt, or machine check. Registers R4 and R5 may be loaded with data pertaining to the exception, interrupt, or machine check. The specific data loaded is described below in conjunction with each exception, interrupt, or machine check; if no specific data is specified, the contents of R4 and R5 are UNPREDICTABLE. After the stack is built, the contents of registers R6 and R7 are UNPREDICTABLE.

The Program Counter value that is saved in the stack frame is:

- For faults, the instruction that encountered the exception
- For traps, the next instruction
- For interrupts and (on a best-effort basis) machine checks, the instruction that would have been issued if the interrupt or machine-check condition had not occurred.

Return from an exception, interrupt, or machine check is done via the CALL_PAL REI instruction, which restores the saved values of PC, PS, and R2..R7. Thus, the CALL_PAL REI instruction:

- For faults, re-executes the faulting instruction
- For traps, executes the next instruction
- For interrupts, executes the instruction that would have been executed if the interrupt had not occurred

- For machine checks, continues execution from the point at which the machine check was taken

Table 14–3 Stack Frame

63	R2	0
	R3	:SP
	R4	:+08
	R5	:+16
	R6	:+24
	R7	:+32
	Program Counter (PC)	:+40
	Processor Status (PS)	:+48
		:+56

14.3 Exceptions

Exception service routines execute in response to exception conditions caused by software. Most exception service routines execute in kernel mode, on the kernel stack; all exception service routines execute at the current processor IPL. Change mode exception routines for CHMU/CHMS/CHME execute in the more privileged of the current mode or the target mode (U/S/E) on the matching stack. Exception service routines are usually coded to avoid exceptions; however, nested exceptions can occur.

Types of Exceptions

There are three types of exceptions:

- A fault is an exception condition that occurs during an instruction and leaves the registers and memory in a consistent state such that elimination of the fault condition and subsequent re-execution of the instruction will give correct results. Faults are not guaranteed to leave the machine in exactly the same state it was in immediately prior to the fault, but rather in a state such that the instruction can be correctly executed if the fault condition is removed. The PC saved in the exception stack frame is the address of the faulting instruction. A CALL_PAL REI instruction to this PC will reexecute the faulting instruction.
- An arithmetic trap is an exception condition that occurs at the completion of the operation that caused the exception. Because several instructions may be in various stages of execution at any time, it is possible for multiple arithmetic traps to occur simultaneously. The PC that is saved in the exception frame on traps is that of the next instruction that would have been issued if the trapping condition(s) had not occurred. This is not necessarily the address of the instruction immediately following the one(s) that encountered the trap condition, and the intervening instructions are collectively called the *trap shadow*. See Section 4.7.7.3, for information.

The intervening instructions may have changed operands or other state used by the instruction(s) encountering the trap condition(s). If such is the case, a CALL_PAL REI instruction to this PC does not reexecute the trapping instruction(s), nor does it

reexecute any intervening instructions; it simply continues execution from the point at which the trap was taken.

In general, it is difficult to fix up results and continue program execution at the point of an arithmetic trap. Software can force a trap to be continued more easily without the need for complicated fixup code. This is accomplished by specifying any valid qualifier combination that includes the /S qualifier with each such instruction and following a set of code-generation restrictions in the code that could cause arithmetic traps, allowing those traps to be completed by an OS completion handler.

The AND of all the exception completion qualifiers for trapping instructions is provided to the OS completion handler in the exception summary SWC bit. If SWC is set, the OS completion handler may find the trigger instruction by scanning backward from the trap PC until each register in the register write mask has been an instruction destination. The trigger instruction is the last instruction in I-stream order to get a trap before the trap shadow. If the SWC bit is clear, no fixup is possible. (The trigger instruction may have been followed by a taken branch, so the trap PC cannot be used to find it.)

- A synchronous trap is an exception condition that occurs at the completion of the operation that caused the exception (or, if the operation can only be partially carried out, at the completion of that part of the operation), and no subsequent instruction is issued before the trap occurs.

Synchronous traps are divided into data alignment traps and all other synchronous traps.

14.3.1 Faults

The six types of faults signal that an instruction or its operands are in some way illegal. These faults are all initiated in kernel mode and push an exception stack frame onto the stack. Upon entry to the exception routine, the saved PC (in the exception stack frame) is the virtual address of the faulting instruction.

The six faults include the Floating Disable Fault described in the next section and five memory management faults.

Memory management faults occur when a virtual address translation encounters an exception condition. This can occur as the result of instruction fetch or during a load or store operation.

Immediately following a memory management fault, register R4 contains the exact virtual address encountering the fault condition.

The register R5 contains the "MM Flag" quadword.

"MM Flag" is set as follows:

0000 0000 0000 0000 ₁₆	for a faulting data read
0000 0000 0000 0001 ₁₆	for a faulting I-fetch operation
8000 0000 0000 0000 ₁₆	for a faulting write operation

The faulting instruction is the instruction whose fetch faulted, or the load, store, or PALcode instruction that encountered the fault condition.

Chapter 11 describes the Alpha memory management architecture in more detail.

14.3.1.1 Floating Disabled Fault

A Floating Disabled Fault is an exception that occurs when an attempt is made to execute a floating-point instruction and the floating-point enable (FEN) bit in the HWPCB is not set.

14.3.1.2 Access Control Violation (ACV) Fault

An ACV fault is a memory management fault that indicates that an attempted access to a virtual address was not allowed in the current mode.

ACV faults usually indicate program errors, but in some cases, such as automatic stack expansion, can indicate implicit operating system functions.

ACV faults take precedence over Translation Not Valid, Fault on Read, Fault on Write, and Fault on Execute faults.

ACV faults take precedence over Translation Not Valid faults so that a malicious user could not degrade system performance by causing spurious page faults to pages for which no access is allowed.

14.3.1.3 Translation Not Valid (TNV)

A TNV fault is a memory management fault that indicates that an attempted access was made to a virtual address whose Page Table Entry (PTE) was not valid.

Software may use TNV faults to implement virtual memory capabilities.

14.3.1.4 Fault on Read (FOR)

An FOR fault is a memory management fault that indicates that an attempted data read access was made to a virtual address whose Page Table Entry (PTE) had the Fault on Read bit set.

As a part of initiating the FOR fault, the processor invalidates the Translation Buffer entry that caused the fault to be generated.

Implementation Note:

This allows an implementation to invalidate entries only from the Data-stream Translation Buffer on Fault on Read faults.

The Translation Buffer may reload and cache the old PTE value between the time the FOR fault invalidates the old value from the Translation Buffer and the time software updates the PTE in memory. Software that depends on the processor-provided invalidate must thus be prepared to take another FOR fault on a page after clearing the page's PTE<FOR> bit. The second fault will invalidate the stale PTE from the Translation Buffer, and the processor cannot load another stale copy. Thus, in the worst case, a multiprocessor system will take an initial FOR fault and then an additional FOR fault on each processor. In practice, even a single repetition is unlikely.

Software may use FOR faults to implement watchpoints, to collect page usage statistics, and to implement execute-only pages.

14.3.1.5 Fault on Write (FOW)

A FOW fault is a memory management fault that indicates that an attempted data write access was made to a virtual address whose Page Table Entry (PTE) had the Fault On Write bit set.

As a part of initiating the FOW fault, the processor invalidates the Translation Buffer entry that caused the fault to be generated.

Implementation Note:

This allows an implementation to invalidate entries only from the Data-stream Translation Buffer on Fault on Write faults.

Note that the Translation Buffer may reload and cache the old PTE value between the time the FOW fault invalidates the old value from the Translation Buffer and the time software updates the PTE in memory. Software that depends on the processor-provided invalidate must thus be prepared to take another FOW fault on a page after clearing the page's PTE<FOW> bit. The second fault will invalidate the stale PTE from the Translation Buffer, and the processor cannot load another stale copy. Thus, in the worst case, a multiprocessor system will take an initial FOW fault and then an additional FOW fault on each processor. In practice, even a single repetition is unlikely.

Software may use FOW faults to maintain modified page information, to implement copy on write and watchpoint capabilities, and to collect page usage statistics.

14.3.1.6 Fault on Execute (FOE)

An FOE fault is a memory management fault that indicates that an attempted instruction stream access was made to a virtual address whose Page Table Entry (PTE) had the Fault On Execute bit set.

As a part of initiating the FOE fault, the processor invalidates the Translation Buffer entry that caused the fault to be generated.

Implementation Note:

This allows an implementation to invalidate entries only from the Instruction-stream Translation Buffer on Fault on Execute faults.

Note that the Translation Buffer may reload and cache the old PTE value between the time the FOE fault invalidates the old value from the Translation Buffer and the time software updates the PTE in memory. Software that depends on the processor-provided invalidate must thus be prepared to take another FOE fault on a page after clearing the page's PTE<FOE> bit. The second fault will invalidate the stale PTE from the Translation Buffer, and the processor cannot load another stale copy. Thus, in the worst case, a multiprocessor system will take an initial FOE fault and then an additional FOE fault on each processor. In practice, even a single repetition is unlikely.

Software may use FOE faults to implement access mode changes and protected entry to kernel mode, to collect page usage statistics, and to detect programming errors that try to execute data.

14.3.2 Arithmetic Traps

An arithmetic trap is an exception that occurs as the result of performing an arithmetic or conversion operation.

If integer register R31 or floating-point register F31 is specified as the destination of an operation that can cause an arithmetic trap, it is UNPREDICTABLE whether the trap will actually occur, even if the operation would definitely produce an exceptional result. If the operation causes an arithmetic trap, the bit that corresponds to R31 or F31 in the Register Write Mask is UNPREDICTABLE.

Arithmetic traps are initiated in kernel mode and push the exception stack frame on the kernel stack. The Register Write Mask is saved in R4, and the Exception Summary parameter is saved in R5. These are described in Section 14.3.2.1.

14.3.2.1 Exception Summary Parameter

The Exception Summary parameter shown in Figure 14–4 and described in Table 14–4 records the various types of arithmetic traps that can occur together. These types of traps are described in subsections below.

Figure 14–4: Exception Summary

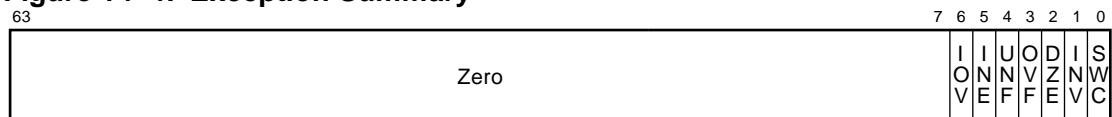


Table 14–4 Exception Summary

Bit	Description
63–7	Zero.
6	Integer Overflow (IOV) An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision.
5	Inexact Result (INE) A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.
4	Underflow (UNF) A floating arithmetic or conversion operation underflowed the destination exponent.
3	Overflow (OVF) A floating arithmetic or conversion operation overflowed the destination exponent.
2	Division by Zero (DZE) An attempt was made to perform a floating divide operation with a divisor of zero.
1	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.
0	Software Completion (SWC) Set when all of the other arithmetic exception bits were set by floating-operate instructions with the /S exception completion qualifier set. See Section 4.7.7.3 for rules about setting the /S qualifier in code that may cause an arithmetic trap, and Section 14.3 for rules about using the SWC bit in a trap handler.

14.3.2.2 Register Write Mask

The Register Write Mask parameter records all registers that were targets of instructions that set the bits in the exception summary register. There is a one-to-one correspondence between bits in the Register Write Mask quadword and the register numbers. The quadword records, starting at bit 0 and proceeding right to left, which of the registers R0 through R31, then F0 through F31, received an exceptional result.

Note:

For a sequence such as:

```
ADDF F1, F2, F3
MULF F4, F5, F3
```

If the add overflows and the multiply does not, the OVF bit is set in the exception summary, and the F3 bit is set in the register mask, even though the overflowed sum in F3 can be overwritten with an in-range product by the time the trap is taken. (This code violates the destination reuse rule for software completion. See Section 4.7.7.3 for the destination reuse rules.)

The PC value saved in the exception stack frame is the virtual address of the next instruction. This is defined as the virtual address of the first instruction not executed after the trap condition was recognized.

14.3.2.3 Invalid Operation (INV) Trap

An INV trap is reported for most floating-point operate instructions with an input operand that is a VAX reserved operand, VAX dirty zero, IEEE NaN, IEEE infinity, or IEEE denormal.

Floating INV traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.

14.3.2.4 Division by Zero (DZE) Trap

A DZE trap is reported when a finite number is divided by zero. Floating DZE traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.

14.3.2.5 Overflow (OVF) Trap

An OVF trap is reported when the destination's largest finite number is exceeded in magnitude by the rounded true result. Floating OVF traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.

14.3.2.6 Underflow (UNF) Trap

A UNF trap is reported when the destination's smallest finite number exceeds in magnitude the non-zero rounded true result. Floating UNF trap enable can be specified in each floating-point operate instruction. If underflow occurs, the result register is written with a true zero.

14.3.2.7 Inexact Result (INE) Trap

An INE trap is reported if the rounded result of an IEEE operation is not exact. INE trap enable can be specified in each IEEE floating-point operate instruction. The unchanged result value is stored in all cases.

14.3.2.8 Integer Overflow (IOV) Trap

An IOV trap is reported for any integer operation whose true result exceeds the destination register size. IOV trap enable can be specified in each arithmetic integer operate instruction and each floating-point convert-to-integer instruction. If integer overflow occurs, the result register is written with the truncated true result.

14.3.3 Synchronous Traps

A synchronous trap is an exception condition that occurs at the completion of the operation that caused the exception (or, if the operation can only be partially carried out, at the completion of that part of the operation), but no successor instruction is allowed to start. All traps that are not arithmetic traps are synchronous traps.

Some synchronous traps are caused by PALcode instructions: BPT, BUGCHK, CHMU, CHMS, CHME, and CHMK. For synchronous traps, the PC saved in the exception stack frame is the address of the instruction immediately following the one causing the trap condition. A CALL_PAL REI instruction to this PC will continue without reexecuting the trapping instruction. The following subsections describe the synchronous traps in detail.

14.3.3.1 Data Alignment Trap

All data must be naturally aligned or an alignment trap may be generated. Natural alignment means that data bytes are on byte boundaries, data words are on word boundaries, data longwords are on longword boundaries, and data quadwords are on quadword boundaries.

A Data Alignment trap is generated by the hardware when an attempt is made to load or store a word, a longword, or a quadword to/from a register using an address that does not have the natural alignment of the particular data reference.

Data Alignment traps are fixed up by the PALcode and are optionally reported to the operating system under the control of the DAT bit. If the bit is zero, the trap will be reported. If the bit is set, after the alignment is corrected, control is returned to the user. In either case, if the PALcode detects a LDx_L or STx_C instruction, no correction is possible and an illegal operand exception is generated.

Note:

In the case of concurrently pending data alignment and arithmetic traps, it is assumed that the arithmetic trap is reported before PALcode data alignment fixup is performed. Otherwise, it would not be possible to back up the PC for the synchronous data alignment trap as required by Section 14.7.4.

The system software is notified via the generation of a kernel mode exception through the Unaligned_Access SCB vector (280₁₆). The virtual address of the unaligned data being accessed is stored in R4. R5 indicates whether the operation was a read or a write (0 = read/load 1 = write/store).

PALcode may write partial results to memory without probing to make sure all writes will succeed when dealing with unaligned store operations.

If a memory management exception condition occurs while reading or writing part of the unaligned data, the appropriate memory management fault is generated.

Software should avoid data misalignment whenever possible since the emulation performance penalty may be as large as 100-to-1.

The Data Alignment trap control bit is included in the HWPCB at offset HWPCB[56], bit 63. In order to change this bit for the currently executing process, the DATFX IPR may be written by using a CALL_PAL MTPR_DATFX instruction. This operation will also update the value in the HWPCB.

14.3.3.2 Other Synchronous Traps

With the traps described in this subsection, the SCB vector quadword is saved in R2 and the SCB parameter quadword is saved in R3. The change mode traps are initiated in the more privileged of the current mode and the target mode, while the other traps are initiated in kernel mode.

14.3.3.2.1 Breakpoint Trap

A Breakpoint trap is an exception that occurs when a CALL_PAL BPT instruction is executed (see Section 10.1.1). Breakpoint traps are intended for use by debuggers and can be used to place breakpoints in a program.

Breakpoint traps are initiated in kernel mode so that system debuggers can capture breakpoint traps that occur while the user is executing system code.

14.3.3.2.2 Bugcheck Trap

A Bugcheck trap is an exception that occurs when a CALL_PAL BUGCHK instruction is executed (see Section 10.1.2). Bugchecks are used to log errors detected by software.

14.3.3.2.3 Illegal Instruction Trap

An Illegal Instruction trap is an exception that occurs when an attempt is made to execute an instruction when:

- It has an opcode that is reserved to Compaq or reserved to PALcode.
- It is a subsetted opcode that requires emulation on the host implementation.
- It is a privileged instruction and the current mode is not kernel.
- It has an unused function code for those opcodes defined as reserved in the Version 5 Alpha architecture specification (May 1992).

14.3.3.2.4 Illegal Operand Trap

An Illegal Operand trap occurs when an attempt is made to execute PALcode with operand values that are illegal or reserved for future use by Compaq. Illegal operands include:

- An invalid combination of bits in the PS restored by the CALL_PAL REI instruction.
- An unaligned operand passed to PALcode.

14.3.3.2.5 Generate Software Trap

A Generate Software trap is an exception that occurs when a CALL_PAL GENTRAP instruction is executed (see Section 10.1.8). The intended use is for low-level compiler-generated code that detects conditions such as divide-by-zero, range errors, subscript bounds, and negative string lengths.

14.3.3.2.6 Change Mode to Kernel Trap

A Change Mode to Kernel trap is an exception that occurs when a CALL_PAL CHMK instruction is executed (see Section 10.1.4). Change Mode to Kernel traps are initiated in kernel mode and push the exception frame on the kernel stack.

14.3.3.2.7 Change Mode to Executive Trap

A Change Mode to Executive trap is an exception that occurs when a CALL_PAL CHME instruction is executed (see Section 10.1.3). Change Mode to Executive traps are initiated in the more privileged of the current mode and Executive mode, and push the exception frame on the target stack.

14.3.3.2.8 Change Mode to Supervisor Trap

A Change Mode to Supervisor trap is an exception that occurs when a CALL_PAL CHMS instruction is executed (see Section 10.1.5). Change Mode to Supervisor traps are initiated in the more privileged of the current mode and supervisor mode, and push the exception frame on the target stack.

14.3.3.2.9 Change Mode to User Trap

A Change Mode to User trap is an exception that occurs when a CALL_PAL CHMU instruction is executed (see Section 10.1.6). Change Mode to User traps are initiated in the more privileged of the current mode and user mode, and push the exception frame on the target stack.

14.4 Interrupts

The processor arbitrates interrupt requests according to priority. When the priority of an interrupt request is higher than the current processor IPL, the processor will raise the IPL and service the interrupt request. The interrupt service routine is entered at the IPL of the interrupting source, in kernel mode, and on the kernel stack. Interrupt requests can come from I/O devices, memory controllers, other processors, or the processor itself.

The priority level of one processor does not affect the priority level of other processors. Thus, in a multiprocessor system, interrupt levels alone cannot be used to synchronize access to shared resources.

Synchronization with other processors in a multiprocessor system involves a combination of raising the IPL and executing an interlocking instruction sequence. Raising the IPL prevents the synchronization sequence itself from being interrupted on a single processor while the interlock sequence guarantees mutual exclusion with other processors. Alternately, one processor can issue explicit interprocessor interrupts (and wait for acknowledgment) to put other processors in a known software state, thus achieving mutual exclusion.

In some implementations, several instructions may be in various stages of execution simultaneously. Before the processor can service an interrupt request, all active instructions must be allowed to complete without exception. Thus, when an exception occurs in a currently active instruction, the exception is initiated and the exception stack frame built immediately before the interrupt is initiated and its stack frame built.

The following events will cause an interrupt:

- Software interrupts — IPL 1 to 15
- Asynchronous System Traps — IPL 2
- Passive Release interrupts — IPL 20 to 23
- I/O Device interrupts — IPL 20 to 23
- Interval Clock interrupt — IPL 22
- Interprocessor interrupt — IPL 22
- Performance Monitor interrupt — IPL 29
- Powerfail interrupt — IPL 30

Interrupts are initiated in kernel mode and push the interrupt stack frame of eight quadwords onto the kernel stack. The PC saved in the interrupt stack frame is the virtual address of the first instruction not executed after the interrupt condition was recognized. A `CALL_PAL REI` instruction to the saved PC/PS will continue execution at the point of interrupt.

Each interrupt source has a separate vector location (offset) within the System Control Block (SCB). (See Section 14.6.) With the exception of I/O device interrupts, each of the above events has a unique fixed vector. I/O device interrupts occupy a range of vectors that can be both statically and dynamically assigned. Upon entry to the interrupt service routine, R2 contains the SCB vector quadword and R3 contains the SCB parameter quadword. For Corrected Error interrupts, R4 optionally locates additional information (see Section 14.5.2).

In order to reduce interrupt overhead, no memory mapping information is changed when an interrupt occurs. Therefore, the instructions, data, and the contents of the interrupt vector for the interrupt service routine must be present in every process at the same virtual address.

Interrupt service routines should follow the discipline of not lowering IPL below their initial level. Lowering IPL in this way could result in an interrupt at an intermediate level, which would cause the stack nesting to be incorrect.

Kernel mode software may need to raise and lower IPL during certain instruction sequences that must synchronize with possible interrupt conditions (such as powerfail). This can be accomplished by specifying the desired IPL and executing a `CALL_PAL MTPR_IPL` instruction or by executing a `CALL_PAL REI` instruction that restores a PS that contains the desired IPL (see Section 10.6.5).

14.4.1 Software Interrupts — IPLs 1 to 15

14.4.1.1 Software Interrupt Summary Register

The architecture provides 15 priority interrupt levels for use by software (level 0 is also available for use by software but interrupts can never occur at this level). The Software Interrupt Summary Register (SISR) stores a mask of pending software interrupts. Bit positions in this mask that contain a 1 correspond to the levels on which software interrupts are pending.

When the processor IPL drops below that of the highest requested software interrupt, a software interrupt is initiated and the corresponding bit in the SISR is cleared.

The SISR is a read-only internal processor register that may be read by kernel mode software by executing a `CALL_PAL MFPR_SISR` instruction (see Section 13.3).

14.4.1.2 Software Interrupt Request Register

The Software Interrupt Request Register (SIRR) is a write-only internal processor register used for making software interrupt requests.

Kernel mode software may request a software interrupt at a particular level by executing a `CALL_PAL MTPR_SIRR` instruction (see Section 13.3).

If the requested interrupt level is greater than the current IPL, the interrupt will occur before the execution of the next instruction. If, however, the requested level is equal to or less than the current processor IPL, the interrupt request will be recorded in the Software Interrupt Summary Register (SISR) and deferred until the processor IPL drops to the appropriate level.

Note that no indication is given if there is already a request at the specified level. Therefore, the respective interrupt service routine must not assume that there is a one-to-one correspondence between interrupts requested and interrupts generated. A valid protocol for generating this correspondence is:

1. The requester places information in a control block and then inserts the control block in a queue associated with the respective software interrupt level.
2. The requester uses `CALL_PAL MTPR_SIRR` to request an interrupt at the appropriate level.
3. When enabling conditions arise, processor HW clears the appropriate SISR bit as part of initiating the software interrupt.
4. The interrupt service routine attempts to remove a control block from the request queue. If there are no control blocks in the queue, the interrupt is dismissed with a `CALL_PAL REI` instruction.
5. If a valid control block is removed from the queue, the requested service is performed and step 3 is repeated.

14.4.2 Asynchronous System Trap — IPL 2

Asynchronous System Traps (ASTs) are a means of notifying a process of events that are not synchronized with its execution, but that must be dealt with in the context of the process. An AST is initiated in kernel mode at IPL 2 when the current mode is less privileged than or equal to a mode for which an AST is pending and not disabled, with `PS<IPL>` less than 2 (see Sections 14.7.6 and 12.3).

There are four separate per-mode SCB vectors, one for each of kernel, executive, supervisor, and user modes.

On encountering an AST, the interrupt stack frame is pushed on the kernel stack. The value of the PC saved in this stack frame is the address of the next instruction to have been executed if the interrupt had not occurred. The SCB vector quadword is saved in R2 and the SCB parameter quadword in R3.

14.4.3 Passive Release Interrupts — IPLs 20 to 23

Passive releases occur when the source of an interrupt granted by a processor cannot be determined. This can happen when the requesting I/O device determines that it no longer requires an interrupt after requesting one or when a previously requested interrupt has already been serviced by another processor in some multiprocessor configurations. The interrupt handler for passive releases executes at the priority level of the interrupt request.

14.4.4 I/O Device Interrupts — IPLs 20 to 23

The architecture provides four priority levels for use by I/O devices. I/O device interrupts are requested when the device encounters a completion, attention, or error condition and the respective interrupt is enabled. See Section 26.3.5 for more information.

14.4.5 Interval Clock Interrupt — IPL 22

The interval clock requests an interrupt periodically.

At least 1000 interval clock interrupts occur per second. An entry in the HWRPB contains the number of interval clock interrupts per second that occur in an actual Alpha implementation, scaled up by 4096, and rounded to a 64-bit integer. (See Section 26.1.)

The accuracy of the interval clock must be at least 50 parts per million (ppm).

Hardware/Software Note:

For example, an interval of 819.2 usec derived from a 10 MHz Ethernet clock and a 13-bit counter is acceptable.

To guarantee software progress, the interval clock interrupt should be no more frequent than the time it takes to do 500 main memory accesses. Over the life of the architecture, this interval may well decrease much more slowly than CPU cycle time decreases.

Other constraints may apply to secure kernel systems.

14.4.6 Interprocessor Interrupt — IPL 22

Interprocessor interrupts are provided to enable operating system software running on one processor to interrupt activity on another processor and cause operating system-dependent actions to be performed.

14.4.6.1 Interprocessor Interrupt Request Register

The Interprocessor Interrupt Request Register (IPIR) is a write-only internal processor register used for making a request to interrupt a specific processor.

Kernel mode software may request to interrupt a particular processor by executing a `CALL_PAL MTPR_IPIR` instruction (see Section 13.3).

If the specified processor is the same as the current processor and the current IPL is less than 22, then the interrupt may be delayed and not initiated before the execution of the next instruction.

Note that, as with software interrupts, no indication is given as to whether there is already an interprocessor interrupt pending when one is requested. Therefore, the interprocessor interrupt service routine must not assume there is a one-to-one correspondence between interrupts requested and interrupts generated. A valid protocol similar to the one for software interrupts for generating this correspondence is:

1. The requester places information in a control block and then inserts the control block in a queue associated with the target processor.
2. The requester uses `CALL_PAL MTPR_IPIR` to request an interprocessor interrupt on the target processor.
3. The interprocessor interrupt service routine on the target processor attempts to remove a control block from its request queue. If there are no control blocks remaining, the interrupt is dismissed with a `CALL_PAL REI` instruction.
4. If a valid control block is removed from the queue, the specified action is performed and step 3 is repeated.

14.4.7 Performance Monitor Interrupts — IPL 29

These interrupts provide some of the support for processor or system performance measurements. The implementation is processor or system specific.

14.4.8 Powerfail Interrupt — IPL 30

If the system power supply backup option permits powerfail recovery, a powerfail interrupt is generated to each processor when power is about to fail. See Section 27.5 for a description of powerfail recovery requirements and for a description of the interactions between system software and the console during system restarts.

In systems in which the backup option maintains only the contents of memory and keeps system time with the `BB_WATCH`, the power supply requests a powerfail interrupt to permit volatile system state to be saved. Prior to dispatching to the powerfail interrupt service routine, PALcode is responsible for saving all system state that is not visible to system software. Such state includes, but is not limited to, processor internal registers and PALcode temporary variables.

PALcode is also responsible for saving the contents of any write-back caches or buffers, including the powerfail interrupt stack frame. System software is responsible for saving all other system state. Such state includes, but is not limited to, processor registers and write-back cache contents. State can be saved by forcing all written data to a backed-up part of the memory subsystem; software may use the `CALL_PAL CFLUSH` instruction.

The powerfail interrupt will not be initiated until the processor IPL drops below 30. Thus, critical code sequences can block the power-down sequence by raising the IPL to 31. Software, however, must take extra care not to lock out the power-down sequence for an extended period of time. The time interval is platform specific.

Explicit state is not provided by the architecture for software to directly determine whether there were outstanding interrupts when powerfail occurred. It is the responsibility of software to leave sufficient information in memory so that it may determine the proper action on power-up.

14.5 Machine Checks

A machine check, or mcheck, indicates that a hardware error condition was detected and may or may not be successfully corrected by hardware or PALcode. Such error conditions can occur either synchronously or asynchronously with respect to instruction execution. There are four types:

1. System Machine Check (IPL 31)

These machine checks are generated by error conditions that are detected asynchronously to processor execution but are not successfully corrected by hardware or PALcode. Examples of system machine check conditions include protocol errors on the processor-memory-interconnect (PMI) and unrecoverable memory errors.

System machine checks are always maskable and deferred until processor IPL drops below IPL 31.

2. Processor Machine Check (IPL 31)

These machine checks indicate that a processor internal error was detected and not successfully corrected by hardware or PALcode. Examples of processor machine check conditions include processor internal cache errors, translation buffer parity errors, or read access to a nonexistent local I/O space location (NXM).

Processor machine checks may be nonmaskable or maskable. If nonmaskable, they are initiated immediately, even if the processor IPL is 31. If maskable, they are deferred until processor IPL drops below IPL 31.

3. System Correctable Machine Check (IPL 20)

These machine checks are generated by error conditions that are detected asynchronously to processor execution and are successfully corrected by hardware or PALcode. Examples of system correctable machine check conditions include single-bit errors within the memory subsystem.

System correctable machine checks are always maskable and deferred until processor IPL drops below IPL 20.

4. Processor Correctable Machine Check (IPL 31)

These machine checks indicate that a processor internal error was detected and successfully corrected by hardware or PALcode. Examples of processor correctable machine check conditions include corrected processor internal cache errors and corrected translation buffer table errors.

Processor correctable machine checks may be nonmaskable or maskable. If nonmaskable, they are initiated immediately, even if the processor IPL is 31. If maskable, they are deferred until processor IPL drops below IPL 31.

Machine checks are initiated in kernel mode, on the kernel stack, and cannot be disabled.

Correctable machine checks permit the pattern and frequency of certain errors to be captured. The delivery of these machine checks to system software can be disabled by setting IPR MCES<4:3>, as described in Section 13.3.9. Note that setting IPR MCES<4:3> does not disable the generation of the machine check or the correction of the error, but rather suppresses the reporting of that correction to system software.

The PC in the machine check stack frame is that of the next instruction that would have issued if the machine check condition had not occurred. This is not necessarily the address of the instruction immediately following the one encountering the error, and intervening instructions may have changed operands or other state used by the instruction encountering the error condition. A CALL_PAL REI instruction to this PC will simply continue execution from the point at which the machine check was taken.

Note:

On machine checks, a meaningful PC is delivered on a best-effort basis. The machine state, processor registers, memory, and I/O devices may be indeterminate.

Machine checks may be deliberately generated by software, such as by probing nonexistent memory during memory sizing or searching for local I/O devices. In such a case, the DRAIN PALcode instruction can be called to force any outstanding machine checks to be taken before continuing.

14.5.1 Software Response

The reaction of system software to machine checks is specific to the characteristics of the processor, platform, and system software. System software must determine if operation should be discontinued on an implementation-specific basis.

To assist system software, PALcode provides a retry flag in the machine check logout frame (see Figure 14-5). If the retry flag is set, the state of the processor and platform hardware has not been compromised; system software operation should be able to continue.

If the retry flag is clear, the state of the processor is either unknown or is known to have been updated during partial execution of one or more instructions. System software operation can continue only after system software determines that the hardware state change permits and/or takes corrective action.

PALcode should take appropriate implementation-specific actions prior to setting the retry flag. PALcode should also attempt to ensure that each encountered error condition generates only one machine check.

Implementation Note:

An important example of using the retry flag is read NXM. Also, a read NXM should not generate both a Processor Machine Check and a System Machine Check.

PALcode sets an internal Machine-Check-In-Progress flag in the Machine Check Error Summary (MCES) register prior to initiating a system or processor machine check. System software must clear that flag to dismiss the machine check. If a second uncorrectable machine check hardware error condition is detected while the flag is set, or if PALcode cannot deliver the machine check, PALcode forces the processor to enter console I/O mode, and subsequent actions, such as processor restart, are taken by the console. The REASON FOR HALT code is "double error abort encountered." See Sections 26.1.3 and 27.5.

Similarly, PALcode sets an internal correctable Machine-Check-In-Progress flag in the Machine Check Error Summary (MCES) register prior to initiating a system-correctable error interrupt or processor-correctable machine check. System software must clear that flag to dis-

miss the condition and permit the reuse of the logout area. If a second correctable hardware error condition is detected while the flag is set, the error is corrected, but not reported. PALcode does not overwrite the logout area and the processor remains in program I/O mode.

14.5.2 Logout Areas

When a hardware error condition is encountered, PALcode optionally builds a logout frame prior to passing control to the machine check service routine. The logout frame is shown in Figure 14–5 and described in Table 14–5. The logout frame is built in the logout area located by the processor’s per-CPU slot in the HWRPB (see Section 26.1).

Figure 14–5: Corrected Error and Machine Check Logout Frame

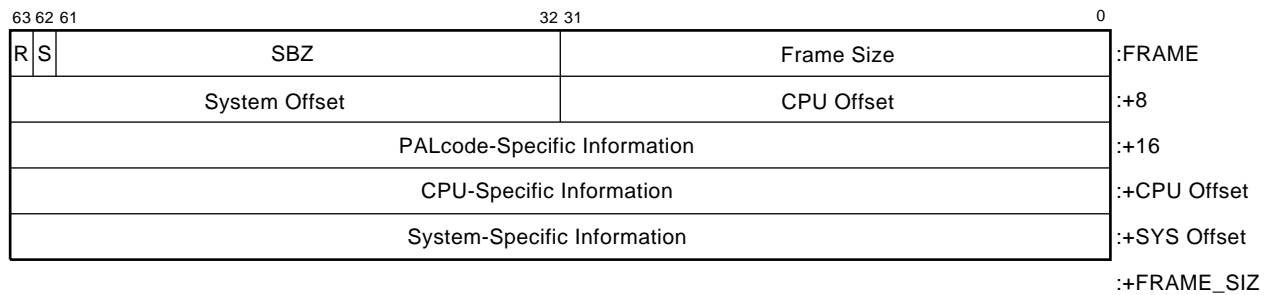


Table 14–5 Corrected Error and Machine Check Logout Frame Fields

Offset	Description								
FRAME	FRAME SIZE — Size in bytes of the logout frame, including the FRAME SIZE longword.								
+04	FRAME FLAGS — Informational flags.								
	<table border="1" style="width: 100%;"> <thead> <tr> <th style="text-align: left;">Bit</th> <th style="text-align: left;">Description</th> </tr> </thead> <tbody> <tr> <td>31</td> <td>RETRY FLAG — Indicates whether execution can be resumed after dismissing this machine check. Set on Corrected Error interrupts; may be set on machine checks.</td> </tr> <tr> <td>30</td> <td>SECOND ERROR FLAG — Indicates that a second correctable error was encountered. Set on Corrected Error interrupts when a correctable error was encountered while the relevant correctable error bit (PCE or SCE) is set in the MCES register. Clear on machine checks.</td> </tr> <tr> <td>29–0</td> <td>SBZ.</td> </tr> </tbody> </table>	Bit	Description	31	RETRY FLAG — Indicates whether execution can be resumed after dismissing this machine check. Set on Corrected Error interrupts; may be set on machine checks.	30	SECOND ERROR FLAG — Indicates that a second correctable error was encountered. Set on Corrected Error interrupts when a correctable error was encountered while the relevant correctable error bit (PCE or SCE) is set in the MCES register. Clear on machine checks.	29–0	SBZ.
Bit	Description								
31	RETRY FLAG — Indicates whether execution can be resumed after dismissing this machine check. Set on Corrected Error interrupts; may be set on machine checks.								
30	SECOND ERROR FLAG — Indicates that a second correctable error was encountered. Set on Corrected Error interrupts when a correctable error was encountered while the relevant correctable error bit (PCE or SCE) is set in the MCES register. Clear on machine checks.								
29–0	SBZ.								
+08	CPU OFFSET — Offset in bytes from the base of the logout frame to the CPU-specific information. If CPU OFFSET is equal to 16, the frame contains no PALcode-specific information. If CPU OFFSET is equal to SYS OFFSET, the frame contains no CPU-specific information.								
+12	SYS OFFSET — Offset in bytes from the base of the logout frame to the system-specific information. If SYS OFFSET is equal to FRAME SIZE, the frame contains no system-specific information.								

Table 14–5 Corrected Error and Machine Check Logout Frame Fields (Continued)

Offset	Description
+16	PALCODE INFORMATION — PALcode-specific logout information.
+CPU OFFSET	CPU INFORMATION — CPU-specific logout information.
+SYS OFFSET	SYS INFORMATION — System platform-specific logout information.

The logout frame is optional; the service routine uses R4 to locate the frame, if any. Upon entry to the service routine, R4 contains the byte offset of the logout frame from the base of the logout area. If no frame was built, R4 contains –1.

14.6 System Control Block

The System Control Block (SCB) specifies the entry points for exception, interrupt, and machine check service routines. The block is from 8K to 32K bytes long, must be page aligned, and must be physically contiguous. The PFN is specified by the value of the System Control Block Base (SCBB) internal register.

The SCB, shown in Figure 14–6, consists of from 512 to 2048 entries, each 16 bytes long. The first eight bytes of an entry, the vector, specify the virtual address of the service routine associated with that entry. The second eight bytes, the parameter, are an arbitrary quadword value to be passed to the service routine.

Table 14–6 System Control Block Summary

Faults	000-0F0
Arithmetic Traps	200-230
Asynchronous System Traps	240-270
Data Alignment Traps	280-3F0
Other Synchronous Traps	400-4F0
Software Interrupts	500-5F0
Processor Hardware Interrupts and Machine Checks	600-6F0
Unused	700-7F0
I/O Hardware Interrupts	800-7FF0

The SCB entries are grouped as follows:

- Faults
- Arithmetic traps
- Asynchronous system traps
- Data alignment trap
- Other synchronous traps
- Processor software interrupts
- Processor hardware interrupts and machine checks

- I/O device interrupts

The first 512 entries (offsets 0000 through 800₁₆) contain all architecturally defined and any statically allocated entries. All remaining SCB entries, if any, are used only for those I/O device interrupt vectors that are assigned dynamically by system software. It is the responsibility of that software to ensure the consistency of the assigned vector and the SCB entry.

14.6.1 SCB Entries for Faults

The exception handler for a fault executes with the IPL unchanged, in kernel mode, on the kernel stack. Table 14–7 lists the SCB entries for faults.

Table 14–7: SCB Entries for Faults

Byte Offset ₁₆	Entry Name
000	Unused
010	Floating Disabled fault
020–070	Unused
080	Access Control Violation fault
090	Translation Not Valid fault
0A0	Fault on Read fault
0B0	Fault on Write fault
0C0	Fault on Execute fault
0A0–0F0	Unused

14.6.2 SCB Entries for Arithmetic Traps

The exception handler for an arithmetic trap executes with the IPL unchanged, in kernel mode, on the kernel stack. Table 14–8 lists the SCB entries for arithmetic traps.

Table 14–8: SCB Entries for Arithmetic Traps

Byte Offset ₁₆	Entry Name
200	Arithmetic Trap
210–230	Unused

14.6.3 SCB Entries for Asynchronous System Traps (ASTs)

The interrupt handler for an asynchronous system trap executes at IPL 2, in kernel mode, on the kernel stack. Table 14–9 lists the SCB entries for asynchronous system traps.

Table 14–9: SCB Entries for Asynchronous System Traps

Byte Offset ₁₆	Entry Name
240	Kernel Mode AST
250	Executive Mode AST
260	Supervisor Mode AST
270	User Mode AST

14.6.4 SCB Entries for Data Alignment Traps

The exception handler for a data alignment trap executes with the IPL unchanged in kernel mode, on the kernel stack. Table 14–10 lists the SCB entries for data alignment traps.

Table 14–10: SCB Entries for Data Alignment Trap

Byte Offset ₁₆	Entry Name
280	Unaligned_Access
290-3F0	Unused

14.6.5 SCB Entries for Other Synchronous Traps

The exception handler for a synchronous trap, other than those described above, executes with the IPL unchanged, in the mode and on the stack indicated below. "MostPriv" indicates that the handler executes in either the original mode or the new mode, whichever is the most privileged. Table 14–11 lists the SCB entries for other synchronous traps.

Table 14–11: SCB Entries for Other Synchronous Traps

Byte Offset ₁₆	Entry Name	Mode
400	Breakpoint Trap	Kernel
410	Bugcheck Trap	Kernel
420	Illegal Instruction Trap	Kernel
430	Illegal Operand Trap	Kernel
440	Generate Software Trap	Kernel
450	Unused	
460	Unused	
470	Unused	
480	Change Mode to Kernel	Kernel
490	Change Mode to Executive	MostPriv

Table 14–11: SCB Entries for Other Synchronous Traps (Continued)

Byte Offset ₁₆	Entry Name	Mode
4A0	Change Mode to Supervisor	MostPriv
4B0	Change Mode to User	Current
4C0–4F0	Reserved for Compaq	

14.6.6 SCB Entries for Processor Software Interrupts

The exception handler for a processor software interrupt executes at the target IPL, in kernel mode, on the kernel stack. Table 14–12 lists the SCB entries for processor software interrupts.

Table 14–12: SCB Entries for Processor Software Interrupts

Byte Offset ₁₆	Entry Name	Target IPL ₁₀
500	Unused	
510	Software interrupt level 1	1
520	Software interrupt level 2	2
530	Software interrupt level 3	3
540	Software interrupt level 4	4
550	Software interrupt level 5	5
560	Software interrupt level 6	6
570	Software interrupt level 7	7
580	Software interrupt level 8	8
590	Software interrupt level 9	9
5A0	Software interrupt level 10	10
5B0	Software interrupt level 11	11
5C0	Software interrupt level 12	12
5D0	Software interrupt level 13	13
5E0	Software interrupt level 14	14
5F0	Software interrupt level 15	15

14.6.7 SCB Entries for Processor Hardware Interrupts and Machine Checks

The interrupt handler for a processor hardware interrupt executes at the target IPL, in kernel mode, on the kernel stack.

The handler for machine checks executes in kernel mode, on the kernel stack. The handler for system-correctable machine checks executes at IPL 20; the handler for all other machine checks executes at IPL 31. Table 14–13 lists the SCB entries for processor hardware interrupts and machine checks.

Table 14–13 SCB Entries for Processor Hardware Interrupts and Machine Checks

Byte Offset ₁₆	Entry Name	Target IPL ₁₀
600	Interval clock interrupt	22
610	Interprocessor interrupt	22
620	System correctable machine check	20
630	Processor correctable machine check	31
640	Powerfail interrupt	30
650	Performance monitor	29
660	System machine check	31
670	Processor machine check	31
680–6E0	Reserved — processor specific	
6F0	Passive release	20-23

Processor-specific SCB entries include those used by console devices (if any) or other peripherals dedicated to system support functions.

14.6.8 SCB Entries for I/O Device Interrupts

The interrupt handler for an I/O device interrupt executes at the target IPL, in kernel mode, on the kernel stack. SCB entries for offsets of 800₁₆ through 7FF0₁₆ are reserved for I/O device interrupts.

14.7 PALcode Support

14.7.1 Stack Writeability

In response to various exceptions, interrupts, and machine checks, PALcode pushes information on the kernel stack. PALcode may write this information without first probing to ensure that all such writes to the kernel stack will succeed. If a memory management exception occurs while pushing information, PALcode forces the processor to enter console I/O mode, and subsequent actions, such as processor restart, are taken by the console. The REASON FOR HALT code is "processor halted due to kernel-stack-not-valid." See Sections 26.1.3 and 27.5.

14.7.2 Stack Residency

The user, supervisor, and executive stacks for the current process do not need to be resident. Software running in kernel mode can bring in or allocate stack pages as TNV faults occur. However, since this activity is taking place in kernel mode, the kernel stack must be fully resident.

When the faults TNV, ACV, FOR, and FOW occur on kernel mode references to the kernel stack, they are considered serious system failures from which recovery is not possible. If any of those faults occur, PALcode forces the processor to enter console I/O mode, and subsequent actions, such as processor restart, are taken by the console. The REASON FOR HALT code is "processor halted due to kernel-stack-not-valid." See Sections 26.1.3 and 27.5.

14.7.3 Stack Alignment

Stacks may have arbitrary byte alignment, but performance may suffer if at least octaword alignment is not maintained by software.

PALcode creates stack frames in response to exceptions and interrupts. Before doing so, the target stack is aligned to a 64-byte boundary by setting the six low bits of the target SP to 000000₂. The previous value of these bits is stored in the SP_ALIGN field of the saved PS in memory, for use by a CALL_PAL REI instruction.

Software-constructed stack frames must be 64-byte aligned and have SP_ALIGN properly set; otherwise, a CALL_PAL REI instruction will take an illegal operand trap.

14.7.4 Initiate Exception or Interrupt or Machine Check

Exceptions, interrupts, and machine checks are initiated by PALcode with interrupts disabled. When an exception, interrupt, or machine check is initiated, the associated SCB vector is read to determine the address of the service routine. PALcode then attempts to push the PC, PS, and R2..R7 onto the target stack. When an interrupt (software or hardware but not AST) is initiated, PS<IP> is set to 1 to indicate an interrupt is in progress. Additional parameters may be passed in R4 and R5 on exceptions and machine checks.

During the attempt to push this information, the exceptions (faults) TNV, ACV, and FOW can occur:

- If any of those faults occur when the target stack is user, supervisor, or executive, then the fault is taken on the kernel stack.
- If any of those faults occur when the target stack is the kernel stack, PALcode forces the processor to enter console I/O mode, and subsequent actions, such as processor restart, are taken by the console. The REASON FOR HALT code is "processor halted due to kernel-stack-not-valid." See Sections 26.1.3 and 27.5.

14.7.5 Initiate Exception or Interrupt or Machine Check Model

```
check_for_exception_or_interrupt_or_mcheck:
    IF NOT {ready_to_initiate_exception OR
           ready_to_initiate_interrupt OR
           ready_to_initiate_mcheck} THEN
        BEGIN
            {fetch next instruction}
            {decode and execute instruction}
        END
    ELSE
```

```

BEGIN
    {wait for instructions in progress to complete}
        ! clear interrupt pending
tmp ← 0
IF {exception pending} THEN
    BEGIN
        {back up implementation specific state if necessary,
         this includes the PC if synchronous trap pending}
        new_ipl ← PS<IPL>
        new_mode ← Kernel
    END

ELSE IF {unmaskable mcheck pending} THEN
    BEGIN
        {back up implementation specific state if necessary}
        {attempt correction if appropriate}
        IF {uncorrectable AND MCE<0> = 1} THEN
            {enter console}
        ELSE IF {uncorrectable} THEN
            new_mode ← Kernel
            new_ipl ← 31
                                ! set mcheck error flag
            MCE<0> ← 1
        ELSE IF {reporting enabled} THEN
            new_mode ← Kernel
            new_ipl ← 31
            MCE<2> ← 1
        END
    END

ELSE IF {data alignment trap} THEN
    new_mode ← Kernel

ELSE IF {synchronous trap} THEN
    CASE {opcode} OF
        {back up implementation specific state if necessary}
        CHME: new_mode ← min(PS<CM>,Executive)
        CHMS: new_mode ← min(PS<CM>,Supervisor)
        CHMU: new_mode ← min(PS<CM>,User)
        otherwise: new_mode ← Kernel
    ENDCASE

ELSE IF {maskable uncorrectable mcheck pending and IPL < 31} THEN
    BEGIN
        {back up implementation specific state if necessary}
        IF {MCE<0> = 1} THEN
            {enter console}
        ELSE
            new_mode ← Kernel
            new_ipl ← 31
            MCE<0> ← 1 ! set mcheck error flag
        END
    END
END

```

```

ELSE IF {interrupt pending} THEN
    new_ipl ← {interrupt source IPL}
    tmp ← 1 ! set interrupt pending
    new_mode ← Kernel

ELSE IF {maskable correctable mcheck pending AND
        reporting enabled} THEN
    new_ipl ← 20
    MCES<1> ← 1
    new_mode ← Kernel
END

IPR_SP[PS<CM>] ← SP
new_sp ← IPR_SP[new_mode]
save_align ← new_sp<5:0>
new_sp<5:0> ← 0

PUSH(PS OR LEFT_SHIFT(save_align,56), old_pc, new_mode)
PUSH(R7, R6, new_mode)
PUSH(R5, R4, new_mode)
PUSH(R3, R2, new_mode)

PS<SW> ← 0
PS<CM> ← new_mode
PS<IP> ← tmp
PS<IPL> ← new_ipl
SP ← new_sp

IF {memory management fault} THEN
    R4 ← VA
    R5 ← MMF
END

IF {data alignment trap} THEN
    R4 ← VA
    R5 ← { 0 if read/load 1 if write/store }
END

IF {mcheck or correctable error interrupt} THEN
    IF {logout frame built}
        R4 ← logout_area_offset
    ELSE
        R4 ← -1
    END
END

IF {arithmetic Trap} THEN
    R4 ← register write mask
    R5 ← exception summary
END

IF {software interrupt} THEN
    SISR ← SISR AND NOT{ 2**{ PRIORITY_ENCODE(SISR) } }
END

```

```

vector ← {exception or interrupt or mcheck SCB offset}

R2 ← (SCBB + vector)
R3 ← (SCBB + vector + 8)
PC ← R2

END

GOTO check_for_exception_or_interrupt_or_mcheck

PROCEDURE PUSH(first, last, mode)
BEGIN
  IF ACCESS(new_sp - 16, mode) THEN
    BEGIN
      (new_sp - 8) ← first
      (new_sp - 16) ← last
      new_sp ← new_sp - 16
    RETURN
    END
  ELSE
    {initiate ACV, TNV, or FOW fault, or
    Kernel Stack Not Valid restart sequence}
  END
END

```

14.7.6 PALcode Interrupt Arbitration

The following sections describe the logic for the interrupt conditions produced by the specified operation.

14.7.6.1 Writing the AST Summary Register

Writing the ASTSR internal processor register (Section 13.3) requests an AST for any of the four processor modes. This operation may request an AST on a formerly inactive level and thus cause an AST interrupt. The logic required to check for this condition is:

```

ASTSR<3:0> ← {ASTSR<3:0> AND R16<3:0>} OR R16<7:4>
IF ASTEN<0> AND ASTSR<0> AND {PS<IPL> LT 2} THEN
  {initiate AST interrupt at IPL 2}

```

14.7.6.2 Writing the AST Enable Register

Writing the ASTEN internal processor register (Section 13.3) enables ASTs for any of the four processor modes. This operation may enable an AST on a formerly inactive level and thus cause an AST interrupt. The logic required to check for this condition is:

```

ASTEN<3:0> ← {ASTEN<3:0> AND R16<3:0>} OR R16<7:4>
IF ASTEN<0> AND ASTSR<0> AND {PS<IPL> LT 2} THEN
  {initiate AST interrupt at IPL 2}

```

14.7.6.3 Writing the IPL Register

Writing the IPL internal processor register (Section 13.3) changes the current IPL. This operation may enable an AST or software interrupt on a formerly inactive level and thus cause an AST or software interrupt. The logic required to check for this condition is:

```

PS<IPL> ← R16<4:0>

! check for software interrupt at level 2..15

IF {RIGHT_SHIFT({SISR AND FFFC16 }, PS<IPL> + 1) NE 0} THEN
  {initiate software interrupt at IPL of high bit set in SISR}

! check for AST

IF ASTEN<0> AND ASTSR<0> AND {PS<IPL> LT 2} THEN
  {initiate AST interrupt at IPL 2}

! check for software interrupt at level 1

IF SISR<1> AND {PS<IPL> EQ 0} THEN
  {initiate software interrupt at IPL 1}

```

14.7.6.4 Writing the Software Interrupt Request Register

Writing the SIRR internal processor register (Section 13.3) requests a software interrupt at one of the 15 software interrupt levels. This operation may cause a formerly inactive level to cause a software interrupt. The logic required to check for this condition is:

```

SISR<level> ← 1
IF level GT PS<IPL> THEN
  {initiate software interrupt at IPL level}

```

14.7.6.4.1 Return from Exception or Interrupt

The CALL_PAL REI instruction (Section 10.1.11) writes both the Current Mode and IPL fields of the PS (see Section 14.2). This may enable a formerly disabled AST or software interrupt to occur. The logic required to check for this condition is:

```

PS ← New PS

! check for software interrupt at level 2..15

IF {RIGHT_SHIFT({SISR AND FFFC16 }, PS<IPL> + 1) NE 0} THEN
  {initiate software interrupt at IPL of high bit set in SISR}

! check for AST

tmp ← NOT LEFT_SHIFT(1110(bin), PS<CM>)
IF {{tmp AND ASTEN AND ASTSR}<3:0> NE 0} AND {PS<IPL> LT 2} THEN
  {initiate AST interrupt at IPL 2}

! check for software interrupt at level 1

IF SISR<1> AND {PS<IPL> EQ 0} THEN
  {initiate software interrupt at IPL 1}

```

14.7.6.5 Swap AST Enable

Swapping the AST enable state for the Current Mode results in writing the ASTEN internal processor register (see Section 13.3). This operation may enable a formerly disabled AST to cause an AST interrupt. The logic required to check for this condition is:

```
R0 ← ZEXT(ASTEN<PS<CM>>)  
ASTEN<PS<CM>> ← R16<0>
```

```
IF ASTEN<PS<CM>> AND ASTSR<PS<CM>> AND {PS<IPL> LT 2} THEN  
  {initiate AST interrupt at IPL 2}
```

14.7.7 Processor State Transition Table

Table 14–14 shows the operations that can produce a state transition and the specific transition produced. For example, if a processor's initial state is supervisor mode, it is not possible for the processor to transition to a program halt condition. A processor can only transition to program halt from kernel mode.

In Table 14–14:

- "REI" increases mode or lowers IPL.
- "MTPR" changes IPL or is a CALL_PAL MTPR_ASTR or CALL_PAL MTPR_ASTEN instruction that causes an interrupt request.
- "Exc" is a state change caused by an exception.
- "Int" is a state change caused by an interrupt.
- "Mcheck" is a state change caused by a machine check

Table 14–14 Processor State Transitions

<u>Initial State:</u>	<u>Final State:</u>				
	User	Super.	Exec.	Kernel	Program Halt
User	CHMU REI	CHMS	CHME	CHMK Exc Int Mcheck SWASTEN	Not Possible
Supervisor	REI	CHMS REI	CHME	CHMK Exc Int Mcheck SWASTEN	Not Possible
Executive	REI	REI	CHME REI	CHMK Exc Int Mcheck SWASTEN	Not Possible
Kernel	REI	REI	REI	CHMK REI Exc Int Mcheck MTPR SWASTEN	HALT

Tru64 UNIX Software (II-B)

The following chapters describe how the Tru64 UNIX operating system relates to the Alpha architecture:

- Chapter 15, Introduction to Tru64 UNIX (II-B)
- Chapter 16, PALcode Instruction Descriptions (II-B)
- Chapter 17, Memory Management (II-B)
- Chapter 18, Process Structure (II-B)
- Chapter 19, Exceptions and Interrupts (II-B)

Chapter 15

Introduction to Tru64 UNIX (II-B)

The goals of this design are to provide a hardware interface between the hardware and Tru64 UNIX that is implementation independent. The interface needs to provide the required abstractions to minimize the impact of different hardware implementations on the operating system. The interface also needs to be low in overhead to support high-performance systems. Finally, the interface needs to support only the features used by Tru64 UNIX.

The register usage in this interface is based on the current calling standard used by Tru64 UNIX. If the calling standard changes, this interface will be changed accordingly. The current calling standard register usage is shown in Table 15–1.

Table 15–1: Tru64 UNIX Register Usage

Register Name	Software Name	Use and Linkage
r0	v0	Used for expression evaluations and to hold integer function results.
r1...r8	t0...t7	Temporary registers; not preserved across procedure calls.
r9...r14	s0...s5	Saved registers; their values must be preserved across procedure calls.
r15	FP or s6	Frame pointer or a saved register.
r16...r21	a0...a5	Argument registers; used to pass the first six integer type arguments; their values are not preserved across procedure calls.
r22...r25	t8...t11	Temporary registers; not preserved across procedure calls.
r26	ra	Contains the return address; used for expression evaluation.
r27	pv or t12	Procedure value or a temporary register.
r28	at	Assembler temporary register; not preserved across procedure calls.
r29	GP	Global pointer.
r30	SP	Stack pointer.
r31	zero	Always has the value 0.

15.1 Programming Model

The programming model of the machine is the combination of the state visible either directly via instructions, or indirectly via actions of the machine. Tables 15–2 and 15–3 define code flow constants, state variables, terms, subroutines, and code flow terms that are used in the rest of the document.

15.1.1 Code Flow Constants and Terms

Tru64 UNIX uses the following constants and terms

Table 15–2 Code Flow Constants and Terms

Term	Meaning and Value
IPL = 2:0	The range 2:0 used in the PS to access the IPL field of the PS (PS <IPL>).
maxCPU	The maximum number of processors in a given system.
mode = 3	Used as a subscript in PS to select current mode (PS <mode>).
opDec	An attempt was made to execute a reserved instruction or execute a privileged instruction in user mode.
pageSize	Size of a page in an implementation in bytes.
vaSize	Size of virtual address in bits in a given implementation.

15.1.2 Machine State Terms

Table 15–3 Machine State Terms

Term	Meaning
ASN	An implementation-dependent size register to hold the current address space number (ASN). The size and existence of ASN is an implementation choice.
entArith <63:0>	The arithmetic trap entry address register. The entArith is an internal processor register that holds the dispatch address on an arithmetic trap. There can be a hardware register for the entArith or the PALcode can use private scratch memory.
entIF <63:0>	The instruction fault or synchronous trap entry address register. The entIF is an internal processor register that holds the dispatch address on an instruction fault or synchronous trap. There can be a hardware register for the entIF or the PALcode can use private scratch memory.
entInt <63:0>	The interrupt entry address register. The entInt is an internal processor register that holds the dispatch address on an interrupt. There can be a hardware register for the entInt or the PALcode can use private scratch memory.
entMM <63:0>	The memory-management fault entry address register. The entMM is an internal processor register that holds the dispatch address on a memory-management fault. There can be a hardware register for the entMM or the PALcode can use private scratch memory.

Table 15–3 Machine State Terms (Continued)

Term	Meaning
entSys <63:0>	The system call entry address register. The entSys is an internal processor register that holds the dispatch address on an callsys instruction. There can be a hardware register for the entSys or the PALcode can use private scratch memory.
entUna <63:0>	The unaligned fault entry address register. The entUna is an internal processor register that holds the dispatch address on an unaligned fault. There can be a hardware register for the entUna or the PALcode can use private scratch memory.
FEN <0>	The floating-point enable register. The FEN is a one-bit register, located at bit 0 of PCB[40], that is used to enable or disable floating-point instructions. If a floating-point instruction is executed with FEN equal to zero, a FEN fault is initiated.
instruction <31:0>	The current instruction being executed. This is a fake register used in the flows to CASE on different instructions.
intr_flag	A per-processor state bit. The intr_flag bit is cleared if that processor executes an rti or retsys instruction.
KGP <63:0>	The kernel global pointer. The KGP is an internal processor register that holds the kernel global pointer that is loaded into R15, the GP, when an exception is initiated. There can be a hardware register for the KGP or the PALcode can use private scratch memory.
KSP <63:0>	The kernel stack pointer. The KSP is an internal processor register that holds the kernel stack pointer while in user mode. There can be a hardware register for the KSP or the storage space in the PCB can be used.
lock_flag <0>	A one-bit register that is used by the load locked and store conditional instructions.
MCES <2:0>	The machine check error summary register. The MCES is a 3-bit register that contains controls for machine check and system-correctable error handling.
PC <63:0>	The program counter. The PC is a pointer to the next instruction in the flows. The low-order two bits of the PC always read as zero and writes to them are ignored.
PCB	The process control block. The PCB holds the state of the process.
PCBB <63:0>	The process control block base address register. The PCBB holds the address of the PCB for the current process.
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>) are an offset, PCC_OFF. PCC_OFF is a value that, when added to PCC_CNT, gives the total PCC register count for this process, modulo 2**32.

Table 15–3 Machine State Terms (Continued)

Term	Meaning
PME <62>	The performance monitoring enable bit. The PME is a one-bit register, located at bit 62 of PCB[40], that alerts any performance monitoring software/hardware in the system that this process is to have its performance monitored. The implementation mechanism for this bit is not specified; it is implementation dependent (IMP).
PS <3:0>	The processor status. The PS is a four-bit register that stores the current mode in bit <3> and stores the three-bit IPL in bits <2:0>. The mode is 0 for kernel and 1 for user.
PTBR <63:0>	The page table base register. The PTBR contains the physical page frame number (PFN) of the highest level page table.
SP <63:0>	Another name for R30. The SP points to the top of the current stack. PALcode only accesses the kernel stack. The kernel stack must be quadword aligned whenever PALcode reads or writes it. If the PALcode accesses the kernel stack and the stack is not aligned, a kernel-stack-not-valid halt is initiated. Although PALcode does not access the user stack, that stack should also be at least quadword aligned for best performance.
SYSPTBR	The system page table physical base register. Contains the page frame number (pfn) of the highest-level page table to be used for system-wide addresses equal to or above the value of the virtual address boundary register. Not saved in a context switch.
sysvalue <63:0>	The system value register. The sysvalue holds the per-processor unique value. There can be a hardware register for the sysvalue register or the storage space in the PALcode scratch memory can be used. The sysvalue register can only be accessed by kernel mode code and there is one sysvalue register per CPU.
unique <63:0>	The process unique value register. The unique register holds the per-process unique value. There can be a hardware register for the unique register or the storage space in the PCB can be used. The unique register can be accessed by both user and kernel code and there is one unique register per process.
USP <63:0>	The user stack pointer. The USP is an internal processor register that holds the user stack pointer while in kernel mode. There can be a hardware register for the USP or the storage space in the PCB can be used.

Table 15–3 Machine State Terms (Continued)

Term	Meaning
VIRBND	The virtual address boundary register. Used to determine which page table physical base register is used. At processor initialization, VIRBND is initialized to a value of -1, which results in all translations using PTBR.
VPTPTR <63:0>	The virtual page table pointer. The VPTPTR holds the virtual address of the first level page table.
whami <63:0>	The processor number of the current processor. This number is in the range 0...maxCPU-1.

PALcode Instruction Descriptions (II–B)

16.1 Unprivileged PALcode Instructions

Table 16–1 lists the Tru64 UNIX PALcode unprivileged instruction mnemonics, names, and the environment from which they can be called.

Table 16–1: Unprivileged PALcode Instructions

Mnemonic	Name	Calling Environment
bpt	Breakpoint trap	Kernel and user modes
bugchk	Bugcheck trap	Kernel and user modes
callsys	System call	User mode
clrfen	Clear floating-point enable	User mode
gentrap	Generate trap	Kernel and user modes
imb	I-stream memory barrier	Kernel and user modes Described in Section 6.7.3.
rdunique	Read unique	Kernel and user modes
urti	Return from user mode trap	User mode
wrunique	Write unique	Kernel and user modes

16.1.1 Breakpoint Trap

Format:

bpt ! PALcode format

Operation:

```
temp ← PS
if (ps<mode> NE 0) then
    USP ← SP          ! Mode is user so switch to kernel
    SP ← KSP
    PS ← 0
endif
SP ← SP - {6 * 8}
(SP+00) ← temp
(SP+08) ← PC
(SP+16) ← GP
(SP+24) ← a0
(SP+32) ← a1
(SP+40) ← a2
a0 ← 0
GP ← KGP
PC ← entIF
```

Exceptions:

Kernel stack not valid

Instruction Mnemonics:

bpt Breakpoint trap

Description:

The breakpoint trap (bpt) instruction switches mode to kernel, builds a stackframe on the kernel stack, loads the GP with the KGP, loads a value of 0 into a0, and dispatches to the breakpoint code pointed to by the entIF register. The registers a1...a2 are UNPREDICTABLE on entry to the trap handler. The saved PC at (SP+08) is the address of the instruction following the trap instruction that caused the trap.

16.1.2 Bugcheck Trap

Format:

bugchk ! PALcode format

Operation:

```
temp ← PS
if (PS<mode> NE 0) then
    USP ← SP          ! Mode is user so switch to kernel
    SP ← KSP
    PS ← 0
endif
SP ← SP - {6 * 8}
(SP+00) ← temp
(SP+08) ← PC
(SP+16) ← GP
(SP+24) ← a0
(SP+32) ← a1
(SP+40) ← a2
a0 ← 1
GP ← KGP
PC ← entIF
```

Exceptions:

Kernel stack not valid

Instruction Mnemonics:

bugchk Bugcheck trap

Description:

The bugcheck trap (bugchk) instruction switches mode to kernel, builds a stackframe on the kernel stack, loads the GP with the KGP, loads a value of 1 into a0, and dispatches to the breakpoint code pointed to by the entIF register. The registers a1...a2 are UNPREDICTABLE on entry to the trap handler. The saved PC at (SP+08) is the address of the instruction following the trap instruction that caused the trap.

16.1.3 System Call

Format:

callsys

! PALcode format

Operation:

```
if (PS<mode> EQ 0) then
    machineCheck
endif
USP ← SP
SP ← KSP
PS ← 0                                ! Mode=kernel
SP ← SP - {6*8}
(SP+00) ← 8                            ! PS of mode=user, IPL=0
(SP+08) ← PC
(SP+08) ← GP
GP ← KGP
PC ← entSys
```

Exceptions:

Machine check – invalid kernel mode callsys
Kernel stack not valid

Instruction Mnemonics:

callsys System call

Description:

The system call (callsys) instruction is supported only from user mode. (Issuing a callsys from kernel mode causes a machine check exception.)

The callsys instruction switches mode to kernel and builds a callsys stack frame. The GP is loaded with the KGP. The exception then dispatches to the system call code pointed to by the entSys register. On entry to the callsys code, the scratch registers t0 and t8...t11 are UNPREDICTABLE.

16.1.4 Clear Floating-Point Enable

Format:

clrfen

! PALcode format

Operation:

$FEN \leftarrow 0$
 $(PCBB+40)\langle 0 \rangle \leftarrow 0$

Exceptions:

None

Instruction Mnemonics:

clrfen Clear floating-point enable

Description:

The clear floating-point enable (clrfen) instruction writes a zero to the floating-point enable register and to the PCB at offset $(PCBB+40)\langle 0 \rangle$. On return from the clrfen instruction, the scratch registers t0 and t8...t11 are UNPREDICTABLE.

16.1.5 Generate Trap

Format:

gentrap ! PALcode format

Operation:

```
temp ← PS
if (PS<mode> NE 0) then
    USP ← SP          ! Mode is user so switch to kernel
    SP ← KSP
    PS ← 0
endif
SP ← SP - {6 * 8}
(SP+00) ← temp
(SP+08) ← PC
(SP+16) ← GP
(SP+24) ← a0
(SP+32) ← a1
(SP+40) ← a2
a0 ← 2
GP ← KGP
PC ← entIF
```

Exceptions:

Kernel stack not valid

Instruction Mnemonics:

gentrap Generate trap

Description:

The generate trap (gentrap) instruction switches mode to kernel, builds a stackframe on the kernel stack, loads the GP with the KGP, loads a value of 2 into a0, and dispatches to the breakpoint code pointed to by the entIF register. The registers a1...a2 are UNPREDICTABLE on entry to the trap handler. The saved PC at (SP+08) is the address of the instruction following the trap instruction that caused the trap.

16.1.6 Read Unique Value

Format:

rdunique

! PALcode format

Operation:

$v0 \leftarrow \text{unique}$

Exceptions:

None

Instruction Mnemonics:

rdunique

Read unique value

Description:

The read unique value (rdunique) instruction returns the process unique value in v0. The write unique value (wrunique) instruction, described in Section 16.1.8, sets the process unique value register.

16.1.7 Return from User Mode Trap

Format:

urti

! PALcode format

Operation:

```
if (PS<mode> EQ 0) then
    {machineCheck}
endif
if (SP<5:0> NE 0)
    {Initiate illegal operand exception}
endif
temppls ← (SP+16)

if (( temppls<mode> EQ 0 ) OR ( temppls<IPL> NE 0 )) then
    {Initiate illegal operand exception}
endif

at      ← (SP+0)
tempssp ← (SP+8)
tempopc ← (SP+24)
GP      ← (SP+32)
a0      ← (SP+40)
a1      ← (SP+48)
a2      ← (SP+56)

intr_flag = 0          ! Clear the interrupt flag
lock_flag = 0          ! Clear the load lock flag

SP      ← tempssp
PC      ← tempopc
```

Exceptions:

- Machine check - invalid kernel mode urti
- Illegal operand
- Translation not valid
- Access violation
- Fault on read

Instruction Mnemonics:

urti Return from user mode trap

Description:

The return from user trap (urti) instruction pops registers (a0...a2, and GP), the new user at, SP, PC, and the PS, from the user stack.

16.1.8 Write Unique Value

Format:

wrunique

! PALcode format

Operation:

unique \leftarrow a0

Exceptions:

None

Instruction Mnemonics:

wrunique

Write unique value

Description:

The write unique value (wrunique) instruction sets the process unique register to the value passed in a0. The read unique value (rdunique) instruction, described in Section 16.1.6, returns the process unique value.

16.2 Privileged PALcode Instructions

The Privileged Tru64 UNIX PALcode instructions (Table 16–2) provide an abstracted interface to control the privileged state of the machine.

Table 16–2: Privileged PALcode Instructions

Mnemonic	Name
cflush	Cache flush
cserve	Console service
draina	Drain aborts. Described in Section 6.7.1.
halt	Halt the processor. Described in Section 6.7.2.
rdmces	Read machine check error summary register
rdps	Read processor status
rdusp	Read user stack pointer
rdval	Read system value
retsys	Return from system call
rti	Return from trap, fault, or interrupt
swpctx	Swap process context
swppal	Swap PALcode image
swpipl	Swap IPL
tbi	TB (translation buffer) invalidate
whami	Who am I
wrasn	Write ASN
wrent	Write system entry address
wrfen	Write floating-point enable
wripir	Write interprocessor interrupt request
wrkgp	Write kernel global pointer
wrmces	Write machine check error summary register
wrperfmon	Performance monitoring function
wrsysptb	Write system page table base
wrusp	Write user stack pointer
wrval	Write system value
wrvirbnd	Write virtual address boundary
wrvptptr	Write virtual page table pointer
wtint	Wait for interrupt

16.2.1 Cache Flush

Format:

cflush

!PALcode format

Operation:

! a0 contains the page frame number (PFN)
! of the page to be flushed

IF PS<mode> EQ 1 THEN
 {Initiate opDec fault}

{Flush page out of cache(s)}

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

cflush

Cache flush

Description:

The cflush instruction may be used to flush an entire physical page specified by the PFN in a0 from any data caches associated with the current processor. All processors must implement this instruction.

On processors that implement a backup power option that maintains only the contents of memory if a powerfail occurs, this instruction is used by the powerfail interrupt handler to force data written by the handler to the battery backed-up main memory. After a cflush, the first subsequent load (on the same processor) to an arbitrary address in the target page is either fetched from physical memory or from the data cache of another processor.

In some multiprocessor systems, cflush is not sufficient to ensure that the data are actually written to memory and not exchanged between processor caches. Additional platform-specific cooperation between the powerfail interrupt handlers executing on each processor may be required.

On systems that implement other backup power options (including none), cflush may return without affecting the data cache contents.

To order cflush properly with respect to preceding writes, an MB instruction is needed before the cflush; to order cflush properly with respect to subsequent reads, an MB instruction is needed after the cflush.

16.2.2 Console Service

Format:

cserve

!PALcode format

Operation:

```
! implementation specific

if PS<mode> EQ 1 then
{initiate opDec fault}

else
{implementation-dependent action}
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

cserve Console service

Description:

This instruction is specific to each PALcode and console implementation and is not intended for operating system use.

16.2.3 Read Machine Check Error Summary

Format:

rdmces

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← MCES
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

rdmces Read machine check error summary

Description:

The read machine check error summary (rdmces) instruction returns the MCES (machine check error summary) register in v0. On return from the rdmces instruction, registers t0 and t8...t11 are UNPREDICTABLE.

16.2.4 Read Processor Status

Format:

rdps

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← PS
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

rdps Read processor status

Description:

The read processor status (rdps) instruction returns the PS in v0. On return from the rdps instruction, registers t0 and t8...t11 are UNPREDICTABLE.

16.2.5 Read User Stack Pointer

Format:

rdusp

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← USP
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

rdusp Read user stack pointer

Description:

The read user stack pointer (rdusp) instruction returns the user stack pointer in v0. The user stack pointer is written by the wrusp instruction, described in Section 16.2.22. On return from the rdusp instruction, registers t0 and t8...t11 are UNPREDICTABLE.

16.2.6 Read System Value

Format:

rdval

!PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← sysvalue
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

rdval Read system value

Description:

The read system value (rdval) instruction returns the sysvalue in v0, allowing access to a 64-bit per-processor value for use by the operating system. On return from the rdval instruction, registers t0 and t8...t11 are UNPREDICTABLE.

16.2.7 Return from System Call

Format:

retsys

! PALcode format

Operation:

```
if {PS<mode> EQ 1} then
    {Initiate opDec fault}
endif
tmp ← (SP+08)
GP ← (SP+16)
KSP ← SP + {6*8}
SP ← USP
intr_flag = 0           ! Clear the interrupt flag
lock_flag = 0          ! Clear the load lock flag
PS ← 8                 ! Mode=user
PC ← tmp
```

Exceptions:

Opcode reserved to Compaq
Kernel stack not valid (halt)

Instruction Mnemonics:

retsys Return from system call

Description:

The return from system call (retsys) instruction pops the return address and the user mode global pointer from the kernel stack. It then saves the kernel stack pointer, sets the mode to user, sets the IPL to zero, and enters the user mode code at the address popped off the stack. On return from the retsys instruction, registers t0 and t8...t11 are UNPREDICTABLE.

16.2.8 Return from Trap, Fault or Interrupt

Format:

rti

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
tempps ← (SP+0)
temppc ← (SP+8)
GP ← (SP+16)
a0 ← (SP+24)
a1 ← (SP+32)
a2 ← (SP+40)
SP ← SP + {6 * 8}
if { tempps<3> EQ 1} then
    KSP ← SP          ! New mode is user
    SP ← USP
    tempps ← 8
endif
intr_flag = 0        ! Clear the interrupt flag
lock_flag = 0       ! Clear the load lock flag
PS ← tempps<3:0>    ! Set new PS
PC ← temppc
```

Exceptions:

Opcode reserved to Compaq

Kernel stack not valid (halt)

Instruction Mnemonics:

rti Return from trap, fault, or interrupt

Description:

The return from fault, trap, or interrupt (rti) instruction pops registers (a0...a2, and GP), the PC, and the PS, from the kernel stack. If the new mode is user, the kernel stack is saved and the user stack is restored.

16.2.9 Swap Process Context

Format:

swpctx

! PALcode format

Operation:

```
if (PS<mode> EQ 1)
    {Initiate opDec fault}
endif
(PCBB) ← SP                ! Save current state
(PCBB+8) ← USP
tmp ← PCC
tmp1 ← tmp<31:0> + tmp<63:32>
(PCBB+24)<31:0> ← tmp1<31:0>
v0 ← PCBB                  ! Return old PCBB
PCBB ← a0                  ! Switch PCBB
SP ← (PCBB)                ! Restore new state
USP ← (PCBB+8)
oldPTBR ← PTBR
PTBR ← (PCBB+16)
tmp1 ← (PCBB+24)
PCC<63:32> ← {tmp1 - tmp}<31:0>
FEN ← (PCBB+40)
if {process unique register implemented} then
    (v0+32) ← unique
    unique ← (PCBB+32)
endif
if {ASN implemented}
    ASN ← tmp1<63:32>
else
    if (oldPTBR NE PTBR)
        {Invalidate all TB entries with ASM=0}
    endif
endif
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

swpctx Swap process context

Description:

The swap process context (swpctx) instruction saves the current process data in the current PCB. Then swpctx switches to the PCB passed in a0 and loads the new process context. The old PCBB is returned in v0.

The process context and the PCB are described in Chapter 12.

On return from the swpctx instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

16.2.10 Swap IPL

Format:

swpipl

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← PS<IPL>
PS<IPL> ← a0<2:0>
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

swpipl Swap IPL

Description:

The swap IPL (swpipl) instruction returns the current value of the PS<IPL> bits in v0 and sets the IPL to the value passed in a0. On return from the swpipl instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

16.2.11 Swap PALcode Image

Format:

swppal

!PALcode format

Operation:

```
! a0 contains the new PALcode identifier
! a1:a5 contain implementation-specific entry parameters
! v0 receives the following status:
!     0 success (PALcode was switched)
!     1 unknown PALcode variant
!     2 known PALcode variant, but PALcode not loaded

if (PS<mode> EQ 1) then
    (Initiate opDec fault)

else
    if {a0 < 256} then
        begin
            if {a0 invalid} then
                v0 ← 1
                {return}
            else if {PALcode not loaded} then
                v0 ← 2
                {return}
            else
                tmp1 ← {PALcode base}
        end
    else
        tmp1 = a0
        {flush instruction cache}
        {invalidate all translation buffers}
        {perform additional PALcode variant-specific initialization}
        {transfer control to PALcode entry at physical address in tmp1}
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

swppal

Swap PALcode image

Description:

The swap PALcode image (swppal) instruction causes the current (active) PALcode to be replaced by the specified new PALcode image. The swppal instruction is intended for use by operating systems only during bootstraps and by consoles during transitions to console I/O mode.

The PALcode descriptor contained in a0 is interpreted as either a PALcode variant or the base physical address of the new PALcode image. If a variant, the PALcode image must have been loaded previously. No PALcode loading occurs as a result of this instruction.

After successful PALcode switching, the register contents are determined by the parameters passed in a1...a5 or are UNPREDICTABLE. A common parameter is the address of a new PCB. In this case, the stack pointer register and PTBR are determined by the contents of that PCB; the contents of other registers such as a0...a5 may be UNPREDICTABLE.

See Section 27.3.2 for information on using this instruction.

16.2.12 TB Invalidate

Format:

tbi ! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
case a0 begin
    1: ! tbisi
        {Invalidate ITB entry for va=a11}
        break;
    2: ! tbi sd
        {Invalidate DTB entry for va=a11}
        break;
    3: ! tbi s
        {Invalidate both ITB and DTB entry for va=a11}
        break;
    -1: ! tbi ap
        {Invalidate all TB entries with ASM=0}
        break;
    -2: ! tbi a
        {Flush all TBs}
        break;
    otherwise:
        break;
endcase
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

tbi TB (translation buffer) invalidate

Description:

The TB invalidate (tbi) instruction removes specified entries from the I and D translation buffers (TBs) when the mapping changes. The tbi instruction removes specific entry types based on a CASE selection of the value passed in register a0. On return from the tbi instruction, registers t0, t8...t11, a0, and a1 are UNPREDICTABLE. See Section 17.7 for information on translation buffers and Section 17.8 for information on address space numbers (ASNs), because ASNs can implicitly modify TB operations.

¹ Operation assumes no behavior modification from ASNs.

16.2.13 Who Am I

Format:

whami

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← whami
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

whami

Who am I

Description:

The who am I (whami) instruction returns the processor number for the current processor in v0. The processor number is in the range 0 to the number of processors minus one (0...max-CPU-1) that can be configured in the system. On return from the whami instruction, registers t0 and t8...t11 are UNPREDICTABLE.

16.2.14 Write ASN

Format:

wrasn

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
```

```
ASN <- a0<31:0>
(PCBB+24)<63:32> <- a0<31:0>
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrasn Write ASN

Description:

The write ASN (wrasn) instruction writes a new ASN. It also writes the value for ASN to the PCB at offset (PCBB+24)<63:32>. On return from the wrasn instruction, registers t0, t8 ...t11, and a0 are UNPREDICTABLE.

16.2.15 Write System Entry Address

Format:

wrent

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
case a1 begin
    0: ! Write the EntInt:
        entInt ← a0
        break;
    1: ! Write the EntArith:
        entArith ← a0
        break;
    2: ! Write the EntMM:
        entMM ← a0
        break;
    3: ! Write the EntIF:
        entIF ← a0
        break;
    4: ! Write the EntUna:
        entUna ← a0
        break;
    5: ! Write the EntSys:
        entSys ← a0
        break;
    otherwise:
        break;
endcase;
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrent Write system entry address

Description:

The write system entry address (wrent) instruction determines the specific system entry point based on a CASE selection of the value passed in register a1. The wrent instruction then sets the virtual address of the specified system entry point to the value passed in a0.

For best performance, all the addresses should be kseg addresses. (See Section 17.1 for a definition of kseg addresses.) On return from the wrent instruction, registers t0, t8...t11, a0, and a1 are UNPREDICTABLE.

16.2.16 Write Floating-Point Enable

Format:

wrfen

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
FEN ← a0<0>
(PCBB+40)<0> ← a0 AND 1
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrfen

Write floating-point enable

Description:

The write floating-point enable (wrfen) instruction writes bit zero of the value passed in a0 to the floating-point enable register. The wrfen instruction also writes the value for FEN to the PCB at offset (PCBB+40)<0>. On return from the wrfen instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

16.2.17 Write Interprocessor Interrupt Request

Format:

wripir

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
IPIR ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wripir Write interprocessor interrupt request

Description:

The write interprocessor interrupt request (wripir) instruction generates an interprocessor interrupt on the processor number passed in register a0. The interrupt request is recorded on the target processor and is initiated when the proper enabling conditions are present. On return from wripir, registers t0, t8...t11, and a0 are UNPREDICTABLE.

Programming Note:

The interrupt need not be initiated before the next instruction is executed on the requesting processor, even if the requesting processor is also the target processor for the request.

16.2.18 Write Kernel Global Pointer

Format:

wrkgp

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
KGP ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrkgp Write kernal global pointer

Description:

The write kernel global pointer (wrkgp) instruction writes the value passed in a0 to the kernel global pointer (KGP) internal register. The KGP is used to load the GP on exceptions. On return from the wrkgp instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

16.2.19 Write Machine Check Error Summary

Format:

wrmces

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
if (a0<0> EQ 1) then MCES<0> ← 0
if (a0<1> EQ 1) then MCES<1> ← 0
if (a0<2> EQ 1) then MCES<2> ← 0
MCES<3> ← a0<3>
MCES<4> ← a0<4>
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrmces Write machine check error summary

Description:

The write machine check error summary (wrmces) instruction 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 enabled bit in the MCES register. On return from the wrmces instruction, registers t0, t8...t11 are UNPREDICTABLE.

16.2.20 Performance Monitoring Function

Format:

wrperfmon ! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
! a0 contains implementation specific input values
! a1 contains implementation specific output values
! v0 may return implementation specific values
! Operations and actions taken are implementation specific
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrperfmon Performance monitoring

Description:

The performance monitoring instruction (wrperfmon) alerts any performance monitoring software/hardware in the system to monitor the performance of this process. The wrperfmon function arguments and actions are platform and chip dependent, and when defined for an implementation, are described in Appendix E.

Registers a0 and a1 contain implementation-specific input values. Implementation-specific values may be returned in register v0. On return from the wrperfmon instruction, registers a0, a1, t0, and t8...t11 are UNPREDICTABLE.

16.2.21 Write System Page Table Base

Format:

wrsysptb

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
```

```
SYSPTBR <- a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrsysptb

Write system page table base

Description:

The write system page table base (wrsysptb) instruction writes the System Page Table Physical Base (SYSPTBR) register. It contains the page frame number (pfn) of the highest level page table to be used for system-wide addresses equal to or above the value of the Virtual Address Boundary Register. The System Page Table and Virtual Address Boundary base registers are described in Section 17.6.

On return from the wrsysptb instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

Note that this register is not context switched.

16.2.22 Write User Stack Pointer

Format:

wrusp

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
USP ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrusp Write user stack pointer

Description:

The write user stack pointer (wrusp) instruction writes the value passed in a0 to the user stack pointer. On return from the wrusp instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

16.2.23 Write System Value

Format:

wrval

!PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
sysvalue ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrval Write system value

Description:

The write system value (wrval) instruction writes the value passed in a0 to a 64-bit system value register. The combination of wrval with the rdval instruction, described in Section 16.2.6, allows access by the operating system to a 64-bit per-processor value. On return from the wrval instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

16.2.24 Write Virtual Address Boundary

Format:

wrvirbnd

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
```

VIRBND ← a0

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrvirbnd

Write virtual address boundary

Description:

The write virtual address boundary (wrvirbnd) instruction writes the virtual address boundary register (VIRBND), used to determine which page table physical base register is used. The System Page Table and Virtual Address Boundary base registers are described in Section 17.6. UNPREDICTABLE operations result if the address is not 64-bit aligned.

On return from the wrvirbnd instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

At processor initialization, VIRBND is initialized to a value of -1, which results in all translations using PTBR. The value in SYSPTBR is thus effectively ignored.

16.2.25 Write Virtual Page Table Pointer

Format:

wrvptptr

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
VPTPTR ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrvptptr Write virtual page table pointer

Description:

The write virtual page table pointer (wrvptptr) instruction writes the pointer passed in a0 to the virtual page table pointer register (VPTPTR). The VPTPTR is described in Section 17.6.2. On return from the wrvptptr instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

16.2.26 Wait For Interrupt

Format:

wtint ! PALcode format

Operation:

! a0 contains the maximum number of interval clock ticks to skip
! v0 receives the number of interval clock ticks actually skipped

```
IF (implemented)
BEGIN
  IF {Implementation supports skipping multiple
      clock interrupts} THEN
    {Ticks_to_skip ← a0}

    {Wait no longer than any non-clock interrupt or the first clock
     interrupt after ticks_to_skip ticks have been skipped}

  IF {Implementation supports skipping multiple
      clock interrupts} THEN
    v0 ← number of interval clock ticks actually skipped
  ELSE
    v0 ← 0
END
ELSE
  v0 ← 0
{return}
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wtint Wait for interrupt

Description:

The wait for interrupt instruction (wtint) 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
- The first clock tick after a specified number of clock ticks has been skipped

The wtint instruction returns in v0 the number of clock ticks that are skipped. The number returned in v0 is zero on hardware platforms that implement this instruction, but where it is not possible to skip clock ticks.

The operating system can specify a full 64-bit integer value in `a0` as the maximum number of interval clock ticks to skip. A value of zero in `a0` causes no clock ticks to be skipped.

Note the following if specifying in `a0` the maximum number of interval clock ticks to skip:

- Adherence to a specified value in `a0` is at the discretion of the PALcode; the PALcode may complete execution of `wtint` and proceed to the next instruction at any time up to the specified maximum, even if no interrupt or interval-clock tick has occurred. That is, `wtint` may return before all requested clock ticks are skipped.
- The PALcode must complete execution of `wtint` if an interrupt occurs or if an interval-clock tick occurs after the requested number of interval-clock ticks has been skipped.

In a multiprocessor environment, only the issuing processor is affected by an issued `wtint` instruction.

The counter, `PCC`, may increment at a lower rate or may stop entirely during `wtint` execution. This side effect is implementation dependent.

Memory Management (II–B)

17.1 Virtual Address Spaces

A virtual address is a 64-bit unsigned integer that specifies a byte location within the virtual address space. Implementations subset the supported address space to one of several sizes, as a function of page size and page table depth. The minimal supported virtual address size is 43 bits. If an implementation supports less than 64-bit virtual addresses, it must check that all the $VA\langle 63:vaSize \rangle$ bits are equal to $VA\langle vaSize-1 \rangle$. This gives two disjoint ranges for valid virtual addresses. For example, for a 43-bit virtual address space, valid virtual address ranges are $0 \dots 3FFFFFFF_{16}$ and $FFFFFC0000000000_{16} \dots FFFFFFFF_{16}$. Access to virtual addresses outside an implementation's valid virtual address range cause an access-violation fault¹.

The virtual address space is divided into three segments: seg0, seg1, and kseg.

The two bits, $va\langle vaSize-1:vaSize-2 \rangle$, select a segment as shown in Table 17–1.

Table 17–1: Virtual Address Space Segments

$VA\langle vaSize-1:vaSize-2 \rangle$	Name	Mapping	Access Control
00	seg0	Mapped via 3 levels of PTEs	Programmed in PTE
01	seg0	Mapped via 2 levels of PTEs	Programmed in PTE
10	kseg	$PA \leftarrow \text{SEXT}(VA\langle vaSize-3:0 \rangle)$	Kernel Read/Write
11	seg1	Mapped via the TB	Programmed in PTE

For kseg, the relocation, sharing, and protection are fixed. The base of kseg is located at $\text{LEFT_SHIFT}(FFFFFC0000000000_{16}, (vaSize-43))$.

For seg0 and seg1, the virtual address space is broken into pages, which are the units of relocation, sharing, and protection. The page size ranges from 8K bytes to 64K bytes. Therefore, system software should allocate regions with differing protection on 64K-byte virtual address boundaries to ensure image compatibility across all Alpha implementations.

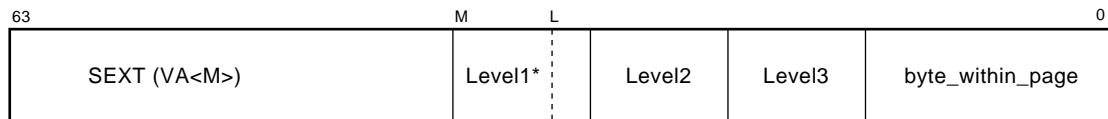
¹ The highest physical address that can be addressed by kseg in 43-bit addressing mode can be extended, under certain circumstances, by an optional 48-bit/43-bit virtual addressing mode, described in Section E.2.1

Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. The operating system controls the virtual-to-physical address mapping tables and saves the inactive (but used) parts of the virtual address space on external storage media.

17.1.1 Segment Seg0 and Seg1 Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. A seg0 or seg1 virtual address consists of three level-number fields and a byte_within_page field, as shown in Figure 17–1.

Figure 17–1: Virtual Address Format



* Level1 <M:L+1> contains SEXT(VA<L>), where L is the highest numbered implemented VA bit.

The byte_within_page field can be either 13, 14, 15, or 16 bits depending on a particular implementation. Thus, the allowable page sizes are 8K bytes, 16K bytes, 32K bytes, and 64K bytes. The low-order bit in each level-number field is 0 and each field is 0...n bits, where for example, n is 9 for an 8K page size.

An implementation may support a smaller virtual address space than the page size allows by including only a subset of low-order bits in Level1. The smaller virtual address space must be at least 43 bits and must be large enough that at least two bits of Level1 are implemented.

The level-number fields are a function of the page size; all page table entries at any given level do not exceed one page. The PFN field in the PTE is always 32 bits wide. Thus, as the page size grows, the virtual and physical address size also grows.

Table 17–2 shows the virtual address options and physical address size (in bits) calculations. The physical address (bits) column is the maximum physical address allowed by the smaller of the kseg size or available physical address bits for a given page size. The available physical address bits is calculated by combining the number of bits in the PFN (always 32) with the number of bits in the byte_within_page field. The kseg segment size is calculated from the virtual address size minus 2.

Table 17–2 Virtual Address Options

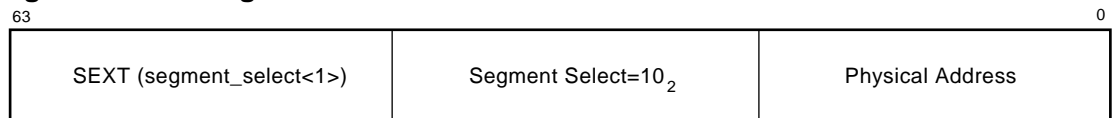
Page Size (bytes)	Byte_within_page (bits)	Level Size (bits)	Virtual Address (bits)	Maximum Physical Address (bits)	Physical Address Limited by
8K	13	10	43	41	kseg
16K	14	11	43–47 ¹	45	kseg
32K	15	12	43–51 ¹	47	seg0/seg1
64K	16	13	44–55 ¹	48	seg0/seg1

¹ Level1 page table might be partially utilized for this page size.

17.1.2 Kseg Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. A kseg virtual address consists of a segment select field with a value of 10_2 and a physical address field. The segment select field is the two bits $va\langle vaSize-1:vaSize-2 \rangle$. The physical address field is $va\langle vaSize-3:0 \rangle$.

Figure 17–2: Kseg Virtual Address Format



17.2 Physical Address Space

Physical addresses are at most $vaSize-2$ bits. This allows all of physical memory to be accessed via kseg. A processor may choose to implement a smaller physical address space by not implementing some number of high-order bits.

The two most significant implemented physical address bits delineate the four regions in the physical address space. Implementations use these bits as appropriate for their systems. For example, in a workstation with a 30-bit physical address space, bit<29> might select between memory and non-memory-like regions, and bit <28> could enable or disable cacheing (see Section 5.2.4).

17.3 Memory Management Control

Memory management is always enabled. Implementations must provide an environment for PALcode to service exceptions and to initialize and boot the processor. For example PALcode might run with I-stream mapping disabled.

17.4 Page Table Entries

The processor uses a quadword page table entry (PTE) to translate seg0 and seg1 virtual addresses to physical addresses. A PTE contains hardware and software control information and the physical page frame number (PFN). A PTE is a quadword with fields as shown in Figure 17–3 and described in Table 17–3.

Figure 17–3 Page Table Entry (PTE)

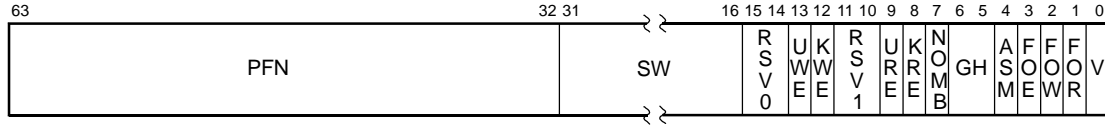


Table 17–3 Page Table Entry (PTE) Bit Summary

Bits	Name	Meaning
63–32	PFN	Page frame number The PFN field always points to a page boundary. If V is set, the PFN is concatenated with the byte_within_page bits of the virtual address to obtain the physical address.
31–16	SW	Reserved for software.
15–14	RSV0	Reserved for hardware; SBZ.
13	UWE	User write enable. Enables writes from user mode. If this bit is 0 and a store is attempted while in user mode, an access-violation fault occurs. This bit is valid even when V=0.
Note:		
If a write enable bit is set and the corresponding read enable bit is not, the operation of the processor is UNDEFINED.		
12	KWE	Kernel write enable. Enables writes from kernel mode. If this bit is 0 and a store is attempted while in kernel mode, an access-violation fault occurs. This bit is valid even when V=0.
11–10	RSV1	Reserved for hardware; SBZ.
9	URE	User read enable. Enables reads from user mode. If this bit is 0 and a load or instruction fetch is attempted while in user mode, an Access Violation occurs. This bit is valid even when V=0.
8	KRE	Kernel read enable. Enables reads from kernel mode. If this bit is 0 and a load or instruction fetch is attempted while in kernel mode, an access-violation fault occurs. This bit is valid even when V=0.
7	NOMB	Translation buffer miss memory barrier. When set, the requirement described in Section 5.6.4.3 is lifted for ensuring that all processors using a newly valid PTE also see any new contents of the related page. This allows the TB-miss code to avoid potentially expensive global synchronization. Software is expected to set this bit on PTEs when it is known that the page contents are already visible to all processors.

Table 17–3 Page Table Entry (PTE) Bit Summary (Continued)

Bits	Name	Meaning																																		
6–5	GH	Granularity hint (GH). Software may set these bits as follows to supply a hint to translation buffer implementations that a block of pages can be treated as a single larger page:																																		
		<table border="1"> <thead> <tr> <th rowspan="2">PTE<6:5></th> <th colspan="4">Page Size Before GH:</th> </tr> <tr> <th>8KB</th> <th>16KB</th> <th>32KB</th> <th>64KB</th> </tr> </thead> <tbody> <tr> <td></td> <td colspan="4">Resulting Page Size:</td> </tr> <tr> <td>00</td> <td>8KB</td> <td>16KB</td> <td>32KB</td> <td>64KB</td> </tr> <tr> <td>01</td> <td>64KB</td> <td>128KB</td> <td>256KB</td> <td>2MB</td> </tr> <tr> <td>10</td> <td>512KB</td> <td>1MB</td> <td>2MB</td> <td>64MB</td> </tr> <tr> <td>11</td> <td>4MB</td> <td>8MB</td> <td>16MB</td> <td>512MB</td> </tr> </tbody> </table>	PTE<6:5>	Page Size Before GH:				8KB	16KB	32KB	64KB		Resulting Page Size:				00	8KB	16KB	32KB	64KB	01	64KB	128KB	256KB	2MB	10	512KB	1MB	2MB	64MB	11	4MB	8MB	16MB	512MB
PTE<6:5>	Page Size Before GH:																																			
	8KB	16KB	32KB	64KB																																
	Resulting Page Size:																																			
00	8KB	16KB	32KB	64KB																																
01	64KB	128KB	256KB	2MB																																
10	512KB	1MB	2MB	64MB																																
11	4MB	8MB	16MB	512MB																																

Notes:

1. The block is a group of physically contiguous pages that are naturally aligned both virtually and physically. Within the block, the PFN field in each PTE must map the correct physical page for the virtual page to which the PTE corresponds.
2. Within the block, all PTEs have the same values for bits <15:0>, that is, protection, fault, granularity, and valid bits.

Hardware may use this hint to map the entire block with a single TB entry.

It is UNPREDICTABLE which PTE values within the block are used if the granularity bits are set inconsistently.

Programming Note:

A granularity hint might be appropriate for a large memory structure such as a frame buffer or nonpaged pool that, in fact, is mapped into contiguous virtual pages with identical protection, fault, and valid bits.

4	ASM	Address space match. When set, this PTE matches all address space numbers. For a given VA, ASM must be set consistently in all processes; otherwise, the address mapping is UNPREDICTABLE.
3	FOE	Fault on execute. When set, a Fault on Execute exception occurs on an attempt to execute any location in the page.

Table 17–3 Page Table Entry (PTE) Bit Summary (Continued)

Bits	Name	Meaning
2	FOW	Fault on write. When set, a Fault on Write exception occurs on an attempt to write any location in the page.
1	FOR	Fault on read. When set, a Fault on Read exception occurs on an attempt to read any location in the page.
0	V	Valid. Indicates the validity of the PFN field. When V is set, the PFN field is valid for use by hardware. When V is clear, the PFN field is reserved for use by software. The V bit does not affect the validity of PTE<15:1> bits.

17.4.1 Changes to Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, the operating system may set or clear the V bit, change the PFN field as pages are moved to and from external storage media, or modify the software bits. The processor hardware never changes PTEs.

Software must guarantee that each PTE is always internally consistent. Changing a PTE one field at a time can cause incorrect system operation, such as setting PTE<V> with one instruction before establishing PTE<PFN> with another. Execution of an interrupt service routine between the two instructions could use an address that would map using the inconsistent PTE. Software can solve this problem by building a complete new PTE in a register and then moving the new PTE to the page table by using an STQ instruction.

Multiprocessing complicates the problem. Another processor could be reading (or even changing) the same PTE that the first processor is changing. Such concurrent access must produce consistent results. Software must use some form of software synchronization to modify PTEs that are already valid. Whenever a processor modifies a valid PTE, it is possible that other processors in a multiprocessor system may have old copies of that PTE in their translation buffer. When software changes a PTE, each processor may use either the old or the new PTE until software performs a TB invalidate on that processor (after which, the processor may use only the new PTE). An example of a case where either the old or new PTE could usefully be used is when the NOMB bit is transitioned from zero to one. Hardware must ensure that aligned quadword reads and writes are atomic operations. Hardware must not cache invalid PTEs (PTEs with the V bit equal to 0) in translation buffers. See Section 17.7 for more information.

17.5 Memory Protection

Memory protection is the function of validating whether a particular type of access is allowed to a specific page from a particular access mode. Access to each page is controlled by a protection code that specifies, for each access mode, whether read or write references are allowed. The processor uses the following to determine whether an intended access is allowed:

- The virtual address, which is used to either select kseg mapping or provide the index into the page tables.

- The intended access type (read or write).
- The current access mode base on processor mode.

For protection checks, the intended access is read for data loads and instruction fetches, and write for data stores.

17.5.1 Processor Access Modes

There are two processor modes, user and kernel. The access mode of a running process is stored in the processor status mode bit (PS<mode>).

17.5.2 Protection Code

Every page in the virtual address space is protected according to its use. A program may be prevented from reading or writing portions of its address space. A protection code associated with each page describes the accessibility of the page for each processor mode.

For seg0 and seg1, the code allows a choice of read or write protection for each processor mode. For each mode, access can be read/write, read-only, or no-access. Read and write accessibility and the protection for each mode are specified independently.

For kseg, the protection code is kernel read/write, user no-access.

17.5.3 Access-Violation Faults

An access-violation memory-management fault occurs if an illegal access is attempted, as determined by the current processor mode and the page's protection.

17.6 Address Translation for Seg0 and Seg1

The page tables can be accessed from physical memory, or (to reduce overhead) can be mapped to a linear region of the virtual address space.

Additionally, an optional reduced page table (RPT) mode is defined, which allows more efficient mapping of very large blocks of memory.

The following sections describe the access methods.

17.6.1 Physical Access for Seg0 and Seg1 PTEs

In systems with Virtual Address Boundary and System Page Table Base registers, the virtual-address is compared against the Virtual Address Boundary register. Lower addresses use the PTBR as a physical page table base; higher or equal addresses use the SYSPTBR register.

Seg0 and seg1 address translation can be performed by accessing entries in a multilevel page table structure. The page table base register (PTBR or SYSPTBR) contains the physical page frame number (PFN) of the highest-level (Level 1) page table.

Bits <Level1> of the virtual address are used to index into the Level 1 page table to obtain the physical PFN of the base of the next level (Level 2) page table. Bits <Level2> of the virtual address are used to index into the Level 2 page table to obtain the physical PFN of the base of the next level (Level 3) page table. Bits <Level3> of the virtual address are used to index the

Level 3 page table to obtain the physical PFN of the page being referenced. The PFN is concatenated with virtual address bits <byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table does not reside in a memory-like region, or does reside in nonexistent memory, the operation of the processor is UNDEFINED.

If all the first- and second-level PTEs are valid, the protection bits are ignored; the protection code in the Level 3 PTE is used to determine accessibility. If a higher-level PTE (numerically, any below Level 3) is invalid, an access-violation fault occurs if the PTE<KRE> equals zero. An access-violation fault on any higher-level PTE implies that all lower-level page tables mapped by that PTE do not exist.

The algorithm to generate a physical address from a seg0 or seg1 virtual address follows:

```

IF {SEXT(VA<vaSize-1>:0) neq VA} THEN
  { initiate access-violation fault}
IF (VIRBND in use) THEN
  IF (VA LTU VIRBND) THEN
    ptbr_value <- PTBR
  ELSE
    ptbr_value <- SYSPTBR
ELSE
  ptbr_value <- PTBR

! Read Physical
level1_pte ←( { ptbr_value * page_size} + { 8 * VA<level1} )

IF level1_pte<v> EQ 0 THEN
  IF level1_pte<KRE> eq 0 THEN
    { initiate access-violation fault}
  ELSE
    { initiate translation-not-valid fault}
! Read physical:
level2_pte " ({level1_pte<PFN> * page_size} + {8 * VA<level2>} )
IF level2_pte<v> EQ 0 THEN
  IF level2_pte<KRE> eq 0 THEN
    { initiate access-violation fault}
  ELSE
    { initiate translation-not-valid fault}
! Read physical:
level3_PTE ← ({level2_pte<PFN> * page_size} + {8 * VA<level3>} )

IF {{{level3_pte<UWE> eq 0}AND {write access} AND {ps<mode> EQ 1}} OR
  {{{level3_pte<URE> eq 0} AND {read access} AND {ps<mode> EQ 1}} OR
  {{{level3_pte<KWE> eq 0}AND {write access} AND {ps<mode> EQ 0}} OR
  {{{level3_pte<KRE> eq 0}AND {read access} AND {ps<mode> EQ 0}}}
  THEN
    {initiate memory-management fault}
  ELSE
    IF level3_pte<v> EQ 0 THEN
      {initiate memory-management fault}

```

```

IF { level3_pte<FOW> eq 1 } AND {write access} THEN
    {initiate memory-management fault}
IF { level3_pte<FOR> eq 1 } AND {read access} THEN
    {initiate memory-management fault}
IF { level3_pte<FOE> eq 1 } AND {execute access} THEN
    {initiate memory-management fault}

```

```
Physical_address ← {level3_pte<PFN> * page_size} OR VA<byte_within_page>
```

17.6.2 Virtual Access for Seg0 or Seg1 PTEs

The page tables can be mapped into a linear region of the virtual address space, reducing the overhead for seg0 and seg1 PTE accesses. If SYSPTBR and VIRBND are implemented, care must be taken to ensure that the Level 3 page tables defined by both PTBR and SYSPTBR are mapped at the same virtual address. This is required so a single VPTPTR can be used regardless of which base register is determined to be used based on the value in VIRBND. (The physical PTE fetch defined in Section 17.6.1 enter the proper mappings into the TB.) The SYSPTBR and VIRBND registers are written by the wrsysptb and wrvirbnd PALcode instructions, described in Sections 16.2.21 and 16.2.24, respectively.

The mapping must be created exactly as follows because PALcode implementations may depend on details of the mapping.

1. Select a $2^{(3*\lg(\text{pageSize}/8)+3)}$ byte-aligned region (an address with $3*\lg(\text{pageSize}/8)+3$ low-order zeros) in the seg0 or seg1 address space.
2. Create a PTE in each of the page tables defined by PTBR and SYSPTBR (if implemented) to map the page tables as follows.

```

PTE = 0                ! Initialize all fields to zero
! Set the PFN to the Level 1 pagetable:
PTE<63:32> = PFN of Level 1 pagetable
PTE<8> = 1             ! Set the kernel read enable bit
PTE<0> = 1             ! Set the valid bit

```

3. Set the page table entry that corresponds to the VPTPTR to the created Level 1 PTE.
4. Set all Level 1 and Level 2 valid PTEs to allow kernel read access. With this setup in place, the algorithm to fetch a seg0 or seg1 PTE is as follows, where pS represents pageSize:

```

tmp ← LEFT_SHIFT (va, {64 - {{lg(pS)* 4} - 9}})
tmp ← RIGHT_SHIFT (tmp, {64 - {{lg(pS)* 4} - 9} + lg(pS)-3})
tmp ← VPTB OR tmp
tmp<2:0> ← 0
level3_PTE ← (tmp)          ! Load PTE using its virtual address

```

5. Set the virtual page table pointer (VPTPTR) with a write virtual page table pointer instruction (wrvp ptr) to the selected value.

The virtual access method is used by PALcode for most TB fills.

Implementation Note:

Assume the following:

- A system with a 52-bit virtual address size.

- VPTB is the index of the Level 1 PTE, which is self-referencing.
- The virtual address is in seg0 or seg1.

For a virtual address B, the address to virtually access the Level 3 PTE is as follows. The double-miss TB fill flow is a three-level flow.

Figure 17–4: Three-Level Page Table Mapping

63	43 42	33 32	23 22	13 12	03 02 0
SEXT (VPTB)	VPTB	B<42:33>	B<32:23>	B<22:13>	0

17.6.3 Reduced Page Table (RPT) Mode

The reduced page table (RPT) mode is an optional extension of 64KB page size mode. A portion of the address space is mapped by one fewer page table levels, allowing each of the entries in the lowest-level page table to map a 512MB page. In implementations that support granularity hints in hardware, applications can use these hints to make more efficient use of the translation buffer. Applications that can use the 512MB granularity hint in 64KB page size mode can use RPT mode for additional benefits.

With the 512MB granularity hint but without RPT, every entry in the Level3 page table maps the same 512MB page. With RPT, that Level3 page table is eliminated entirely, and the Level2 PTE that would normally point to that Level3 page table is used to directly map the 512MB page.

Therefore, in an RPT region, there is elimination of redundant page table pages and compression of page table space. The compressed PTEs are more likely to fit in hardware caches. If there is locality of reference, a new PTE that is needed to satisfy a mapping is more likely to be present in the cache. Additionally, a single TB entry that maps the VA of the lowest-level page table now allows access to PTEs mapping 4 TB, rather than 512 MB, of memory.

In order to use RPT mode, the feature must be available and enabled in the implementation, and:

- Use the 64KB page size.
- Every L2 PTE in the reduced page table region must have $PTE\langle GH \rangle = 11_2$, that is, a 512MB page size.
- The PFN field of the PTE must refer to a 512 MB aligned page.
- The RPT region is selected by using VAs such that $VA\langle vaSize-1:vaSize-2 \rangle = 01_2$.

17.6.3.1 Physical Access for Page Table Entries in Reduced Page Table Mode

Physical address translation is performed by accessing entries in a two-level page table structure. The Page Table Base Register (PTBR) contains the physical Page Frame Number (PFN) of the highest-level (Level1) page table.

In systems that implement the Virtual Address Boundary register (VIRBND), the System Page Table Base Register (SYSPTBR) contains the PFN of an alternate highest-level page table. In such systems, the virtual address to be translated is compared against the address stored in VIRBND. Translations of Level2 addresses begin with the PFN in PTBR as the highest-level

page table. Translations of Level1 addresses use the PFN in SYSPTBR as the highest-level page table. The VIRBND and SYSPTBR registers are described in Sections 13.3.24 and 13.3.18, respectively.

Level1 is the highest-level page table. Bits <Level1> of the virtual address are used to index into the Level1 page table to obtain the physical PFN of the base of the next level (Level2) page table. Bits <Level2> of the virtual address are used to index into the Level2 page table to obtain the physical PFN of the page being referenced. The PFN is concatenated with virtual address bits <byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table resides in I/O space, or in nonexistent memory, the operation of the processor is UNDEFINED.

If the Level1 PTE is valid, the protection bits are ignored; the protection code in the Level2 PTE is used to determine accessibility. If a Level1 PTE is invalid, an access-violation fault occurs if the PTE<KRE> equals zero. An Access-Violation fault on any Level1 PTE implies that all Level2 page tables mapped by that PTE do not exist.

The algorithm to generate a physical address from a virtual address follows:

```

IF {SEXT(VA<vaSize-1>:0) neq VA} THEN
  { initiate access-violation fault}
IF (VIRBND in use) THEN
  IF (VA LTU VIRBND) THEN
    ptbr_value <- PTBR
  ELSE
    ptbr_value <- SYSPTBR
ELSE
  ptbr_value <- PTBR

! Read Physical
level1_pte ←( { ptbr_value * page_size} + { 8 * VA<level1} )

IF level1_pte<v> EQ 0 THEN
  IF level1_pte<KRE> eq 0 THEN
    { initiate access-violation fault}
  ELSE
    { initiate translation-not-valid fault}
! Read physical:
level2_pte " ({level1_pte<PFN> * page_size} + {8 * VA<level2>} )

IF {{{level2_pte<UWE> eq 0}AND {write access} AND {ps<mode> EQ 1}} OR
  {{{level2_pte<URE> eq 0} AND {read access} AND {ps<mode> EQ 1}} OR
  {{{level2_pte<KWE> eq 0}AND {write access} AND {ps<mode> EQ 0}} OR
  {{{level2_pte<KRE> eq 0}AND {read access} AND {ps<mode> EQ 0}}}
  THEN
    {initiate memory-management fault}
  ELSE
    IF level2_pte<v> EQ 0 THEN
      {initiate memory-management fault}

IF { level2_pte<FOW> eq 1} AND {write access} THEN
  {initiate memory-management fault}
IF { level2_pte<FOR> eq 1} AND {read access} THEN
  {initiate memory-management fault}

```

```

IF { level2_pte<FOE> eq 1 } AND {execute access} THEN
    {initiate memory-management fault}
Physical_Address ← {level2_pte<PFN> * page_size} OR VA<byte_within_RPT_page1>

```

17.6.3.2 Virtual Access for Page Table Entries in Reduced Page Table Mode

To reduce overhead associated with the address translation in a multilevel page table structure, the page tables are mapped into a linear region of the virtual address space. The virtual address of the base of the page table structure is set on a system-wide basis and is contained in the VPTB IPR.

When a native mode DTB or ITB miss occurs, it is desirable that the TBMISSE flow attempt to load the lowest-level PTE by using a single virtual load instruction without regard to whether the missing VA is mapped by two levels (RPT) or three levels of page table. (See Section E.2.2 for the 21364 implementation.)

17.7 Translation Buffer

In order to save actual memory references when repeatedly referencing the same pages, hardware implementations include a translation buffer to remember successful virtual address translations and page states.

When the process context is changed, a new value is loaded into the address space number (ASN) internal processor register with a swap process context (swpctx) instruction. This causes address translations for pages with PTE<ASM> clear to be invalidated on a processor that does not implement address space numbers.

Additionally, when the software changes any part (except the software field) of a valid PTE, it must also execute a tbi instruction. The entire translation buffer can be invalidated by tbia, and all ASM=0 entries can be invalidated by tbiap. The translation buffer must not store invalid PTEs. Therefore, the software is not required to invalidate translation buffer entries when making changes for PTEs that are already invalid. Changes to PTE<NOMB> are also an exception to this requirement. This bit only has an effect when a PTE is loaded into the translation buffer. Thus, there is no need to invalidate the TB when the bit changes.

After software changes a valid first-, or second-level PTE, software must flush the translation for the corresponding page in the virtual page table. Then software must flush the translations of all valid pages mapped by that page. In the case of a change to a first-level PTE, this action must be taken through a second iteration.

17.8 Address Space Numbers

The Alpha architecture allows a processor to optionally implement address space numbers (process tags) to reduce the need for invalidation of cached address translations for process-specific addresses when a context switch occurs. The supported address space number (ASN) range is 0...MAX_ASN; MAX_ASN is provided in the HWRPB MAX_ASN field.

¹ byte_within_RPT_page contains those bits that would have been VA<Level3>, concatenated with the VA<byte_within_page> field for 64KB page table mode.

The address space number for the current process is loaded by software in the address space number (ASN) with a swpctx instruction. ASNs are processor specific and the hardware makes no attempt to maintain coherency across multiple processors. In a multiprocessor system, software is responsible for ensuring the consistency of TB entries for processes that might be rescheduled on different processors.

Systems that support ASNs should have MAX_ASN in the range 13...65535. The number of ASNs should be determined by the market a system is targeting.

Programming Note:

System software should not assume that the number of ASNs is a power of two. This allows hardware, for example, to use N TB tag bits to encode $(2^{**N})-3$ ASN values, one value for ASM=1 PTEs, and one for invalid.

There are several possible ways of using ASNs that result from several complications in a multiprocessor system. Consider the case where a process that executed on processor-1 is rescheduled on processor-2. If a page is deleted or its protection is changed, the TB in processor-1 has stale data.

- One solution is to send an interprocessor interrupt to all the processors on which this process could have run and cause them to invalidate the changed PTE. That results in significant overhead in a system with several processors.
- Another solution is to have software invalidate all TB entries for a process on a new processor before it can begin execution, if the process executed on another processor during its previous execution. This ensures the deletion of possibly stale TB entries on the new processor.
- A third solution is to assign a new ASN whenever a process is run on a processor that is not the same as the last processor on which it ran.

17.9 Memory-Management Faults

On a memory-management fault, the fault code (MMCSR) is passed in a1 to specify the type of fault encountered, as shown in Table 17-4.

Table 17-4: Memory-Management Fault Type Codes

Fault	MMCSR Value
Translation not valid	0
Access-violation	1
Fault on read	2
Fault on execute	3
Fault on write	4

- A translation-not-valid fault is taken when a read or write reference is attempted through an invalid PTE in a zero (if one exists), first, second, or third-level page table.

- An access-violation (ACV) fault is taken under the following circumstances:
 - An ACV fault is taken on a reference to a seg0 or seg1 address when the protection field of the third-level PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode.
 - An ACV fault is taken if the KRE bit is a zero in an invalid first-, or second-level PTE. An access-violation fault is generated for any access to a kseg address when the mode is user (PS<mode> EQ 1).
 - For reduced page table regions:
 - An ACV fault is taken when the protection field of the Level2 PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode.
 - An ACV fault is also taken if the KRE bit is zero in an invalid Level1 PTE.
- A fault-on-read (FOR) fault occurs when a read is attempted with PTE<FOR> set.
- A fault-on-execute (FOE) fault occurs when an instruction fetch is attempted with PTE<FOE> set.
- A fault-on-write (FOW) fault occurs when a write is attempted with PTE<FOW> set.

Process Structure (II–B)

18.1 Process Definition

A process is a single thread of execution. It is the basic entity that can be scheduled and is executed by the processor. A process consists of an address space and both software and hardware context. The hardware context of a process is defined by the following:

- Thirty integer registers (excludes R31 and SP)
- Thirty-one floating-point registers (excludes F31)
- The program counter (PC)
- The two per-process stack pointers (USP/KSP)
- The processor status (PS)
- The address space number (ASN)
- The charged process cycles
- The page table base register (PTBR)
- The process unique value (unique)
- The floating-point enable register (FEN)
- The performance monitoring enable bit (PME)

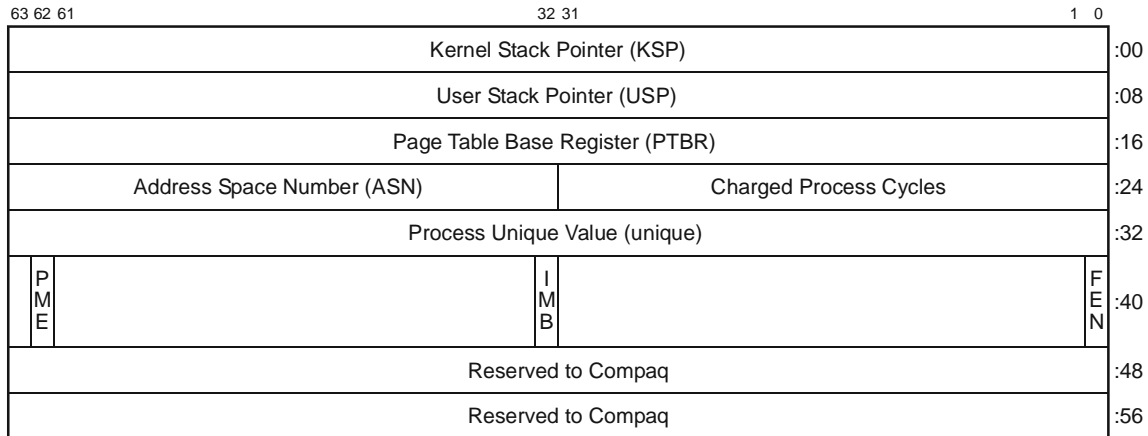
This information must be loaded if a process is to execute.

While a process is executing, some of its hardware context is being updated in the internal registers. When a process is not being executed, its hardware context is stored in memory in a software structure called the process control block (PCB). Saving the process context in the PCB and loading new values from another PCB for a new context is called context switching. Context switching occurs as one process after another is scheduled for execution.

18.2 Process Control Block (PCB)

As shown in Figure 18–1, the PCB holds the state of a process.

Figure 18–1 Process Control Block (PCB)



The contents of the PCB are loaded and saved by the swap process context (swpctx) instruction. The PCB must be quadword aligned and lie within a single page of physical memory. It should be 64-byte aligned for best performance.

The PCB for the current process is specified by the process control block base address register (PCBB); see Table 15–3.

The swap privileged context instruction (swpctx) saves the privileged context of the current process into the PCB specified by PCBB, loads a new value into PCBB, and then loads the privileged context of the new process into the appropriate hardware registers.

The new value loaded into PCBB, as well as the contents of the PCB, must satisfy certain constraints or an UNDEFINED operation results:

1. The physical address loaded into PCBB must be quadword aligned and describes eight contiguous quadwords that are in a memory-like region (see Section 5.2.4).
2. The value of PTBR must be the page frame number (PFN) of an existent page that is in a memory-like region.

It is the responsibility of the operating system to save and load the non-privileged part of the hardware context.

The swpctx instruction returns ownership of the current PCB to operating system software and passes ownership of the new PCB from the operating system to the processor. Any attempt to write a PCB while ownership resides with the processor has UNDEFINED results. If the PCB is read while ownership resides with the processor, it is UNPREDICTABLE whether the original or an updated value of a field is read. The processor is free to update a PCB field at any time. The decision as to whether or not a field is updated is made individually for each field.

The charged process cycles is the total number of PCC register counts that are charged to the process (modulo $2^{*}32$). When a process context is loaded by the swpctx instructions, the contents of the PCC count field (PCC_CNT) is subtracted from the contents of PCB[24]<31:0> and the result is written to the PCC offset field (PCC_OFF):

$PCC\langle 63:32 \rangle \leftarrow (PCB[24]\langle 31:0 \rangle - PCC\langle 31:0 \rangle)$

When a process context is saved by the swpctx instruction, the charged process cycles is computed by performing an unsigned add of $PCC\langle 63:32 \rangle$ and $PCC\langle 31:0 \rangle$. That value is written to $PCB[24]\langle 31:0 \rangle$.

Software Programming Note:

The following example returns in R0 the current PCC register count (modulo $2^{**}32$) for a process. Notice the care taken not to cause an unwanted sign extension.

```
RPCC    R0                ; Read the processor 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
```

If ASNs are not implemented, the ASN field is not read or written by PALcode.

The process unique value is that value used in support of multithread implementations. The value is stored in the PCB when the process is not active. When the process is active, the value may be cached in hardware internal storage or kept in the PCB only.

The FEN bit reflects the setting of the FEN IPR.

The IMB bit records that an IMB was issued in user mode.

Setting the PME bit alerts any performance hardware or software in the system to monitor the performance of this process.

Kernel mode code must use the rdusp/wrusp instructions to access the USP. Kernel mode code can read the PTBR, the ASN, the FEN, and the PME for the current process from the PCB. The unique value can be accessed with the rdunique and wrunique instructions.

Exceptions and Interrupts (II–B)

19.1 Introduction

At certain times during the operation of a system, events within the system require the execution of software outside the explicit flow of control. When such an event occurs, an Alpha processor forces a change in control flow from that indicated by the current instruction stream. The notification process for such an event is either an exception or an interrupt.

19.1.1 Exceptions

Exceptions occur primarily in relation to the currently executing process. Exception service routines execute in response to exception conditions caused by software. All exception service routines execute in kernel mode on the kernel stack. Exception conditions consist of faults, arithmetic traps, and synchronous traps:

- A fault occurs during an instruction and leaves the registers and memory in a consistent state such that elimination of the fault condition and subsequent reexecution of the instruction gives correct results. Faults are not guaranteed to leave the machine in exactly the same state it was in immediately prior to the fault, but rather in a state such that the instruction can be correctly executed if the fault condition is removed. The PC saved in the exception stack frame is the address of the faulting instruction. An *rti* instruction to that PC reexecutes the faulting instruction.
- An arithmetic trap occurs at the completion of the operation that caused the exception. Since several instructions may be in various stages of execution at any point in time, it is possible for multiple arithmetic traps to occur simultaneously.

The PC that is saved in the exception frame on traps is that of the next instruction that would have been issued if the trapping conditions had not occurred. However, that PC is not necessarily the address of the instruction immediately following the instruction that encountered the trap condition, and the intervening instructions are collectively called the *trap shadow*. See Section 4.7.7.3 for information.

The intervening instructions may have changed operands or other state used by the instructions encountering the trap conditions. If such is the case, an *rti* instruction to that PC does not reexecute the trapping instructions, nor does it reexecute any intervening instructions; it simply continues execution from the point at which the trap was taken.

In general, it is difficult to fix up results and continue program execution at the point of an arithmetic trap. Software can force a trap to be continued more easily without the need for complicated fixup code. This is accomplished by specifying any valid

qualifier combination that includes the /S qualifier with each such instruction and following a set of code-generation restrictions in the code that could cause arithmetic traps, allowing those traps to be completed by an OS completion handler.

The AND of all the exception completion qualifiers for trapping instructions is provided to the OS completion handler in the exception summary SWC bit. If SWC is set, a completion handler may find the trigger instruction by scanning backward from the trap PC until each register in the register write mask has been an instruction destination. The trigger instruction is the last instruction in I-stream order to get a trap before the trap shadow. If the SWC bit is clear, no fixup is possible.

- A synchronous trap occurs at the completion of the operation that caused the exception. No instructions can be issued between the completion of the operation that caused the exception and the trap.

19.1.2 Interrupts

The processor arbitrates interrupt requests. When the interrupt priority level (IPL) of an outstanding interrupt is greater than the current IPL, the processor raises IPL to the level of the interrupt and dispatches to entInt, the interrupt entry to the OS. Interrupts are serviced in kernel mode on the kernel stack. Interrupts can come from one of five sources: interprocessor interrupts, I/O devices, the clock, performance counters, or machine checks.

19.2 Processor Status

The processor status (PS) is a four-bit register that contains the current mode (PS<mode>) in bit <3> and a three-bit interrupt priority level (PS<IPL>) in bits <2...0>. The PS<mode> bit is zero for kernel mode and one for user mode. The PS<IPL> bits are always zero if the mode is user and can be zero to 7 if the mode is kernel. The PS is changed when an interrupt or exception is initiated and by the rti, retsys, and swpipl instructions.

The uses of the PS values are shown in Table 19–1.

Table 19–1: Processor Status Summary

PS<mode>	PS<IPL>	Mode	Use
1	0	User	User software
0	0	Kernel	System software
0	1	Kernel	System software
0	2	Kernel	System software
0	3	Kernel	Low priority device interrupts
0	4	Kernel	High priority device interrupts
0	5	Kernel	Clock, and interprocessor interrupts
0	6	Kernel	Real-time devices
0	6	Kernel	Correctable error reporting
0	7	Kernel	Machine checks

19.3 Stack Frames

There are three types of system entries: entries for the `callsys` instruction from user mode, entries for exceptions and interrupts from kernel mode, and entries for interrupts from user mode.

Those three types of system entries use one of two stack frame layouts, as follows.

Entries for the `callsys` instruction from user mode, and entries for exceptions and interrupts from kernel mode use the same stack frame layout, as shown in Figure 19–1. The stack frame contains space for the PC, the PS, the saved GP, and the saved registers `a0`, `a1`, `a2`. On entry, the `SP` points to the saved PS.

The `callsys` entry saves the PC, the PS, and the GP. The exception and interrupt entries save the PC, the PS, the GP, and also save the registers `a0...a2`.

Figure 19–1 Stack Frame Layout for `callsys` and `rti`

63		0
	PS	:00
	PC	:08
	GP	:16
	a0	:24
	a1	:32
	a2	:40

Entries for interrupts from user mode use the stack frame layout as shown in Figure 19–2. The stack frame must be aligned on a 64-byte boundary and contains the registers, `at`, `SP`, `PS`, `PC`, `GP`, and saved registers `a0`, `a1`, and `a2`.

Figure 19–2 Stack Frame Layout for `urti`

63		0
	at	:00
	SP	:08
	PS	:16
	PC	:24
	GP	:32
	a0	:40
	a1	:48
	a2	:56

19.4 System Entry Addresses

All system entries are in kernel mode. The interrupt priority PS bits (PS<IPL>) are set as shown in the following table. The system entry point address is set by the wrent instruction, as described in Section 16.2.15.

Table 19–2 Entry Point Address Registers

Entry Point	Value in a0	Value in a1	Value in a2	PS<IPL>
entArith	Exception summary	Register mask	UNPREDICTABLE	Unchanged
entIF	Fault or trap type code	UNPREDICTABLE	UNPREDICTABLE	Unchanged
entInt	Interrupt type	Vector	Interrupt parameter	Priority of interrupt
entMM	VA	MMCSR	Cause	Unchanged
entSys	p0	p1	p2	Unchanged
entUna	VA	Opcode	Src/Dst	Unchanged

19.4.1 System Entry Arithmetic Trap (entArith)

The arithmetic trap entry, entArith, is called when an arithmetic trap occurs. On entry, a0 contains the exception summary register and a1 contains the exception register write mask. Section 19.4.1.1 describes the exception summary register and Section 19.4.1.2 describes the register write mask.

19.4.1.1 Exception Summary Register

The exception summary register, shown in Figure 19–3 and described in Table 19–3, records the various types of arithmetic exceptions that can occur together.

Figure 19–3 Exception Summary Register



Table 19–3 Exception Summary Register Bit Definitions

Bit	Description
63–7	Zero.
6	<p>Integer overflow (IOV)</p> <p>An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision.</p> <p>An IOV trap is reported for any integer operation whose true result exceeds the destination register size. Integer overflow trap enable can be specified in each arithmetic integer operate instruction and each floating-point convert-to-integer instruction. If integer overflow occurs, the result register is written with the truncated true result.</p>
5	<p>Inexact result (INE)</p> <p>A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.</p> <p>An INE trap is reported if the rounded result of an IEEE operation is not exact. Inexact result trap enable can be specified in each IEEE floating-point operate instruction. The rounded result value is stored in all cases.</p>
4	<p>Underflow (UNF)</p> <p>A floating arithmetic or conversion operation underflowed the destination exponent.</p> <p>An UNF trap is reported when the destination’s smallest finite number exceeds in magnitude the non-zero rounded true result. Floating underflow trap enable can be specified in each floating-point operate instruction. If underflow occurs, the result register is written with a true zero.</p>
3	<p>Overflow (OVF)</p> <p>A floating arithmetic or conversion operation overflowed the destination exponent.</p> <p>An OVF trap is reported when the destination’s largest finite number is exceeded in magnitude by the rounded true result. Floating overflow traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.</p>

Table 19–3 Exception Summary Register Bit Definitions (Continued)

Bit	Description
2	<p>Division by zero (DZE)</p> <p>An attempt was made to perform a floating divide operation with a divisor of zero.</p> <p>A DZE trap is reported when a finite number is divided by zero. Floating divide by zero traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.</p>
1	<p>Invalid operation (INV)</p> <p>An attempt was made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values were illegal.</p> <p>An INV trap is reported for most floating-point operate instructions with an input operand that is an IEEE NaN, IEEE infinity, or IEEE denormal.</p> <p>Floating invalid operation traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.</p>
0	<p>Software completion (SWC)</p> <p>Is set when all of the other arithmetic exception bits were set by floating-operate instructions with the /S qualifier set. See Section 4.7.7.3 for rules about setting the /S qualifier in code that may cause an arithmetic trap, and Section 19.1.1 for rules about using the SWC bit in a trap handler.</p>

19.4.1.2 Exception Register Write Mask

The exception register write mask parameter records all registers that were targets of instructions that set the bits in the exception summary register. There is a one-to-one correspondence between bits in the register write mask quadword and the register numbers. The quadword, starting at bit 0 and proceeding right to left, records which of the registers r0 through r31, then f0 through f31, received an exceptional result.

Note:

For a sequence such as:

```

ADDF    F1, F2, F3
MULF    F4, F5, F3

```

if the add overflows and the multiply does not, the OVF bit is set in the exception summary, and the F3 bit is set in the register mask, even though the overflowed sum in F3 can be overwritten with an in-range product by the time the trap is taken. (This code violates the destination reuse rule for exception completion. See Section 4.7.7.3 for the destination reuse rules.)

The PC value saved in the exception stack frame is the virtual address of the next instruction. This is defined as the virtual address of the first instruction not executed after the trap condition was recognized.

19.4.2 System Entry Instruction Fault (entIF)

The instruction fault or synchronous trap entry is called for bpt, bugchk, gentrap, and opDec

synchronous traps, and for a FEN fault (floating-point instruction when the floating-point unit is disabled, FEN EQ 0). On entry, a0 contains a 0 for a bpt, a 1 for bugchk, a 2 for gentrap, a 3 for FEN fault, and a 4 for opDec. No additional data is passed in a1...a2. The saved PC at (SP+00) is the address of the instruction that caused the fault for FEN faults. The saved PC at (SP+00) is the address of the instruction after the instruction that caused the bpt, bugchk, gentrap, and opDec synchronous traps.

19.4.3 System Entry Hardware Interrupts (entInt)

The interrupt entry is called to service a hardware interrupt or a machine check. Table 19–4 shows what is passed in a0...a2 and the PS<IPL> setting for various interrupts.

Table 19–4 System Entry Hardware Interrupts

Entry Type	Value in a0	Value in a1	Value in a2	PS<IPL>
Interprocessor interrupt	0	UNPREDICTABLE	UNPREDICTABLE	5
Clock	1	UNPREDICTABLE	UNPREDICTABLE	5
Correctable error	2	Interrupt vector	Pointer to Logout Area	7
Machine check	2	Interrupt vector	Pointer to Logout Area	7
I/O device interrupt	3	Interrupt vector	UNPREDICTABLE	Level of device
Performance counter	4	Interrupt vector	UNPREDICTABLE	6

On entry to the hardware interrupt routine, the IPL has been set to the level of the interrupt. For hardware interrupts, register a1 contains a platform-specific interrupt vector. That platform-specific interrupt vector is typically the same value as the SCB offset value that would be returned if the platform was running OpenVMS PALcode.

For a correctable error or machine check interrupt, a1 contains a platform-specific interrupt vector and a2 contains the kseg address of the platform-specific logout area. The interrupt vector value and logout area format are typically the same as those used by the platform when running OpenVMS PALcode.

The machine check error summary (MCES) register, shown in Figure 19–4 and described in Table 19–5, records the correctable error and machine check interrupts in progress.

Figure 19–4 Machine Check Error Status (MCES) Register

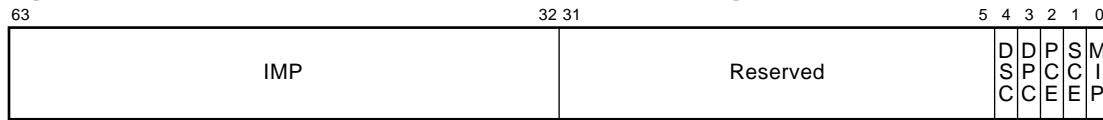


Table 19–5 Machine Check Error Status (MCES) Register Bit Definitions

Bit	Symbol	Description
63–32		IMP.
31–5		Reserved.
4	DSC	Disable system correctable error in progress. Set to disable system correctable error reporting.
3	DPC	Disable processor correctable error in progress. Set to disable processor correctable error reporting.
2	PCE	Processor correctable error in progress. Set when a processor correctable error is detected. Should be cleared by the processor correctable error handler when the logout frame may be reused.
1	SCE	System correctable error in progress. Set when a system correctable error is detected. Should be cleared by the system correctable error handler when the logout frame may be reused.
0	MIP	Machine check in progress. Set when a machine check occurs. Must be cleared by the machine check handler when a subsequent machine check can be handled. Used to detect double machine checks.

The MIP flag in the MCES register is set prior to invoking the machine check handler. If the MIP flag is set when a machine check is being initiated, a double machine check halt is initiated instead. The machine check handler needs to clear the MIP flag when it can handle a new machine check.

Similarly, the SCE or PCE flag in the MCES register is set prior to invoking the appropriate correctable error handler. That error handler should clear the appropriate correctable error in progress when the logout area can be reused by hardware or PALcode. PALcode does not overwrite the logout area.

Correctable processor or system error reporting may be suppressed by setting the respective DPC or DSC flag in the MCES register. When the DPC or DSC flag is set, the corresponding error is corrected, but no correctable error interrupt is generated.

19.4.4 System Entry MM Fault (entMM)

The memory-management fault entry is called when a memory management exception occurs. On entry, a0 contains the faulting virtual address and a1 contains the MMCSR (see Section 17.9). On entry, a2 is set to a minus one (–1) for an instruction fetch fault, to a plus one (+1) for a fault caused by a store instruction, or to a 0 for a fault caused by a load instruction.

19.4.5 System Entry Call System (entSys)

The system call entry is called when a callsys instruction is executed in user mode. On entry, only registers (t8...t11) have been modified. The PC+4 of the callsys instruction, the user global pointer, and the current PS are saved on the kernel stack. Additional space for a0...a2 is allocated. After completion of the system service routine, the kernel code executes a CALL_PAL retsys instruction.

19.4.6 System Entry Unaligned Access (entUna)

The unaligned access entry is called when a load or store access is not aligned. On entry, a0 contains the faulting virtual address, a1 contains the zero extended six-bit opcode (bits <31:26>) of the faulting instruction, and a2 contains the zero extended data source or destination register number (bits <25:21>) of the faulting instruction.

19.5 PALcode Support

19.5.1 Stack Writeability and Alignment

PALcode only accesses the kernel stack. Any PALcode accesses to the kernel stack that would produce a memory-management fault will result in a kernel-stack-not-valid halt. The stack pointer must always point to a quadword-aligned address. If the kernel stack is not quadword aligned on a PALcode access, a kernel-stack-not-valid halt is initiated.

Alpha Linux Software (II-C)

The following chapters describe how the Alpha Linux operating system relates to the Alpha architecture:

- Chapter 20, Introduction to Alpha Linux (II-C)
- Chapter 21, PALcode Instruction Descriptions (II-C)
- Chapter 22, Memory Management (II-C)
- Chapter 23, Process Structure (II-C)
- Chapter 24, Exceptions and Interrupts (II-C)

Introduction to Alpha Linux (II–C)

The goals of this design are to provide a hardware interface between the hardware and Alpha Linux that is implementation independent. The interface needs to provide the required abstractions to minimize the impact of different hardware implementations on the operating system. The interface also needs to be low in overhead to support high-performance systems. Finally, the interface needs to support only the features used by Alpha Linux.

The register usage in this interface is based on the current calling standard used by Alpha Linux. If the calling standard changes, this interface will be changed accordingly. The current calling standard register usage is shown in Table 20–1.

Table 20–1 Alpha Linux Register Usage

Register Name	Software Name	Use and Linkage
r0	v0	Used for expression evaluations and to hold integer function results.
r1...r8	t0...t7	Temporary registers; not preserved across procedure calls.
r9...r14	s0...s5	Saved registers; their values must be preserved across procedure calls.
r15	FP or s6	Frame pointer or a saved register.
r16...r21	a0...a5	Argument registers; used to pass the first six integer type arguments; their values are not preserved across procedure calls.
r22...r25	t8...t11	Temporary registers; not preserved across procedure calls.
r26	ra	Contains the return address; used for expression evaluation.
r27	pv or t12	Procedure value or a temporary register.
r28	at	Assembler temporary register; not preserved across procedure calls.
r29	GP	Global pointer.
r30	SP	Stack pointer.
r31	zero	Always has the value 0.

20.1 Programming Model

The programming model of the machine is the combination of the state visible either directly via instructions, or indirectly via actions of the machine. Tables 20–2 and 20–3 define code flow constants, state variables, terms, subroutines, and code flow terms that are used in the rest of the document.

20.1.1 Code Flow Constants and Terms

Alpha Linux uses the following constants and terms.

Table 20–2 Code Flow Constants and Terms

Term	Meaning and value
IPL = 2:0	The range 2:0 used in the PS to access the IPL field of the PS (PS <IPL>).
maxCPU	The maximum number of processors in a given system.
mode = 3	Used as a subscript in PS to select current mode (PS <mode>).
opDec	An attempt was made to execute a reserved instruction or execute a privileged instruction in user mode.
pageSize	Size of a page in an implementation in bytes.
vaSize	Size of virtual address in bits in a given implementation.

20.1.2 Machine State Terms

Table 20–3 Machine State Terms

Term	Meaning
ASN	An implementation-dependent size register to hold the current address space number (ASN). The size and existence of ASN is an implementation choice.
entArith <63:0>	The arithmetic trap entry address register. The entArith is an internal processor register that holds the dispatch address on an arithmetic trap. There can be a hardware register for the entArith or the PALcode can use private scratch memory.
entIF <63:0>	The instruction fault or synchronous trap entry address register. The entIF is an internal processor register that holds the dispatch address on an instruction fault or synchronous trap. There can be a hardware register for the entIF or the PALcode can use private scratch memory.
entInt <63:0>	The interrupt entry address register. The entInt is an internal processor register that holds the dispatch address on an interrupt. There can be a hardware register for the entInt or the PALcode can use private scratch memory.
entMM <63:0>	The memory-management fault entry address register. The entMM is an internal processor register that holds the dispatch address on a memory-management fault. There can be a hardware register for the entMM or the PALcode can use private scratch memory.

Table 20–3 Machine State Terms (Continued)

Term	Meaning
entSys <63:0>	The system call entry address register. The entSys is an internal processor register that holds the dispatch address on a callsys instruction. There can be a hardware register for the entSys or the PALcode can use private scratch memory.
entUna <63:0>	The unaligned fault entry address register. The entUna is an internal processor register that holds the dispatch address on an unaligned fault. There can be a hardware register for the entUna or the PALcode can use private scratch memory.
FEN <0>	The floating-point enable register. The FEN is a one-bit register, located at bit 0 of PCB[40], that is used to enable or disable floating-point instructions. If a floating-point instruction is executed with FEN equal to zero, a FEN fault is initiated.
instruction <31:0>	The current instruction being executed. This is a fake register used in the flows to CASE on different instructions.
intr_flag	A per-processor state bit. The intr_flag bit is cleared if that processor executes an rti or retsys instruction.
KGP <63:0>	The kernel global pointer. The KGP is an internal processor register that holds the kernel global pointer that is loaded into R15, the GP, when an exception is initiated. There can be a hardware register for the KGP or the PALcode can use private scratch memory.
KSP <63:0>	The kernel stack pointer. The KSP is an internal processor register that holds the kernel stack pointer while in user mode. There can be a hardware register for the KSP or the storage space in the PCB can be used.
lock_flag <0>	A one-bit register that is used by the load locked and store conditional instructions.
MCES <2:0>	The machine check error summary register. The MCES is a 3-bit register that contains controls for machine check and system-correctable error handling.
PC <63:0>	The program counter. The PC is a pointer to the next instruction in the flows. The low-order two bits of the PC always read as zero and writes to them are ignored.
PCB	The process control block. The PCB holds the state of the process.
PCBB <63:0>	The process control block base address register. The PCBB holds the address of the PCB for the current process.
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>) are an offset, PCC_OFF. PCC_OFF is a value that, when added to PCC_CNT, gives the total PCC register count for this process, modulo 2**32.

Table 20–3 Machine State Terms (Continued)

Term	Meaning
PME <62>	The performance monitoring enable bit. The PME is a one-bit register, located at bit 62 of PCB[40], that alerts any performance monitoring software/hardware in the system that this process is to have its performance monitored. The implementation mechanism for this bit is not specified; it is implementation dependent (IMP).
PS <3:0>	The processor status. The PS is a four-bit register that stores the current mode in bit <3> and stores the three-bit IPL in bits <2:0>. The mode is 0 for kernel and 1 for user.
PTBR <63:0>	The page table base register. The PTBR contains the physical page frame number (PFN) of the highest level page table.
SP <63:0>	Another name for R30. The SP points to the top of the current stack. PALcode only accesses the kernel stack. The kernel stack must be quadword aligned whenever PALcode reads or writes it. If the PALcode accesses the kernel stack and the stack is not aligned, a kernel-stack-not-valid halt is initiated. Although PALcode does not access the user stack, that stack should also be at least quadword aligned for best performance.
SYSPTBR	The system page table physical base register. Contains the page frame number (pfn) of the highest-level page table to be used for system-wide addresses equal to or above the value of the virtual address boundary register. Not saved in a context switch.
sysvalue <63:0>	The system value register. The sysvalue holds the per-processor unique value. There can be a hardware register for the sysvalue register or the storage space in the PALcode scratch memory can be used. The sysvalue register can only be accessed by kernel mode code and there is one sysvalue register per CPU.
unique <63:0>	The process unique value register. The unique register holds the per-process unique value. There can be a hardware register for the unique register or the storage space in the PCB can be used. The unique register can be accessed by both user and kernel code and there is one unique register per process.
USP <63:0>	The user stack pointer. The USP is an internal processor register that holds the user stack pointer while in kernel mode. There can be a hardware register for the USP or the storage space in the PCB can be used.

Table 20–3 Machine State Terms (Continued)

Term	Meaning
VIRBND	The virtual address boundary register. Used to determine which page table physical base register is used. At processor initialization, VIRBND is initialized to a value of -1, which results in all translations using PTBR.
VPTPTR <63:0>	The virtual page table pointer. The VPTPTR holds the virtual address of the first level page table.
whami <63:0>	The processor number of the current processor. This number is in the range 0...maxCPU-1.

PALcode Instruction Descriptions (II–C)

21.1 Unprivileged PALcode Instructions

Table 21–1 lists the Alpha Linux PALcode unprivileged instruction mnemonics, names, and the environment from which they can be called.

Table 21–1: Unprivileged PALcode Instructions

Mnemonic	Name	Calling Environment
bpt	Breakpoint trap	Kernel and user modes
bugchk	Bugcheck trap	Kernel and user modes
callsys	System call	User mode
clrfen	Clear floating-point enable	User mode
gentrap	Generate trap	Kernel and user modes
imb	I-stream memory barrier	Kernel and user modes Described in Section 6.7.3.
rdunique	Read unique	Kernel and user modes
wrunique	Write unique	Kernel and user modes

21.1.1 Breakpoint Trap

Format:

bpt ! PALcode format

Operation:

```
temp ← PS
if (ps<mode> NE 0) then
    USP ← SP          ! Mode is user so switch to kernel
    SP ← KSP
    PS ← 0
endif
SP ← SP - {6 * 8}
(SP+00) ← temp
(SP+08) ← PC
(SP+16) ← GP
(SP+24) ← a0
(SP+32) ← a1
(SP+40) ← a2
a0 ← 0
GP ← KGP
PC ← entIF
```

Exceptions:

Kernel stack not valid

Instruction Mnemonics:

bpt Breakpoint trap

Description:

The breakpoint trap (bpt) instruction switches mode to kernel, builds a stackframe on the kernel stack, loads the GP with the KGP, loads a value of 0 into a0, and dispatches to the breakpoint code pointed to by the entIF register. The registers a1...a2 are UNPREDICTABLE on entry to the trap handler. The saved PC at (SP+08) is the address of the instruction following the trap instruction that caused the trap.

21.1.2 Bugcheck Trap

Format:

bugchk ! PALcode format

Operation:

```
temp ← PS
if (PS<mode> NE 0) then
    USP ← SP           ! Mode is user so switch to kernel
    SP ← KSP
    PS ← 0
endif
SP ← SP - {6 * 8}
(SP+00) ← temp
(SP+08) ← PC
(SP+16) ← GP
(SP+24) ← a0
(SP+32) ← a1
(SP+40) ← a2
a0 ← 1
GP ← KGP
PC ← entIF
```

Exceptions:

Kernel stack not valid

Instruction Mnemonics:

bugchk Bugcheck trap

Description:

The bugcheck trap (bugchk) instruction switches mode to kernel, builds a stackframe on the kernel stack, loads the GP with the KGP, loads a value of 1 into a0, and dispatches to the breakpoint code pointed to by the entIF register. The registers a1...a2 are UNPREDICTABLE on entry to the trap handler. The saved PC at (SP+08) is the address of the instruction following the trap instruction that caused the trap.

21.1.3 System Call

Format:

callsys

! PALcode format

Operation:

```
if (PS<mode> EQ 0) then
    machineCheck
endif
USP ← SP
SP ← KSP
PS ← 0                                ! Mode=kernel
SP ← SP - {6*8}
(SP+00) ← 8                            ! PS of mode=user, IPL=0
(SP+08) ← PC
(SP+08) ← GP
GP ← KGP
PC ← entSys
```

Exceptions:

Machine check – invalid kernel mode callsys
Kernel stack not valid

Instruction Mnemonics:

callsys System call

Description:

The system call (callsys) instruction is supported only from user mode. (Issuing a callsys from kernel mode causes a machine check exception.)

The callsys instruction switches mode to kernel and builds a callsys stack frame. The GP is loaded with the KGP. The exception then dispatches to the system call code pointed to by the entSys register. On entry to the callsys code, the scratch registers t0 and t8...t11 are UNPREDICTABLE.

21.1.4 Clear Floating-Point Enable

Format:

clrfen

! PALcode format

Operation:

$FEN \leftarrow 0$
 $(PCBB+40)\langle 0 \rangle \leftarrow 0$

Exceptions:

None

Instruction Mnemonics:

clrfen Clear floating-point enable

Description:

The clear floating-point enable (clrfen) instruction writes a zero to the floating-point enable register and to the PCB at offset $(PCBB+40)\langle 0 \rangle$. On return from the clrfen instruction, the scratch registers t0 and t8...t11 are UNPREDICTABLE.

21.1.5 Generate Trap

Format:

```
gentrap                                     ! PALcode format
```

Operation:

```
temp ← PS
if (PS<mode> NE 0) then
    USP ← SP          ! Mode is user so switch to kernel
    SP ← KSP
    PS ← 0
endif
SP ← SP - {6 * 8}
(SP+00) ← temp
(SP+08) ← PC
(SP+16) ← GP
(SP+24) ← a0
(SP+32) ← a1
(SP+40) ← a2
a0 ← 2
GP ← KGP
PC ← entIF
```

Exceptions:

Kernel stack not valid

Instruction Mnemonics:

```
gentrap          Generate trap
```

Description:

The generate trap (`gentrap`) instruction switches mode to kernel, builds a stackframe on the kernel stack, loads the GP with the KGP, loads a value of 2 into a0, and dispatches to the breakpoint code pointed to by the entIF register. The registers a1...a2 are UNPREDICTABLE on entry to the trap handler. The saved PC at (SP+08) is the address of the instruction following the trap instruction that caused the trap.

21.1.6 Read Unique Value

Format:

rdunique

! PALcode format

Operation:

$v0 \leftarrow \text{unique}$

Exceptions:

None

Instruction Mnemonics:

rdunique

Read unique value

Description:

The read unique value (rdunique) instruction returns the process unique value in v0. The write unique value (wrunique) instruction, described in Section 21.1.7, sets the process unique value register.

21.1.7 Write Unique Value

Format:

wrunique

! PALcode format

Operation:

unique \leftarrow a0

Exceptions:

None

Instruction Mnemonics:

wrunique

Write unique value

Description:

The write unique value (wrunique) instruction sets the process unique register to the value passed in a0. The read unique value (rdunique) instruction, described in Section 21.1.6, returns the process unique value.

21.2 Privileged PALcode Instructions

The Privileged Alpha Linux PALcode instructions (Table 21–2) provide an abstracted interface to control the privileged state of the machine.

Table 21–2: Privileged PALcode Instructions

Mnemonic	Name
cflush	Cache flush
cserve	Console service
draina	Drain aborts. Described in Section 6.7.1.
halt	Halt the processor. Described in Section 6.7.2.
rdmces	Read machine check error summary register
rdps	Read processor status
rdusp	Read user stack pointer
rdval	Read system value
retsys	Return from system call
rti	Return from trap, fault, or interrupt
swpctx	Swap process context
swpipl	Swap IPL
swppal	Swap PALcode image
tbi	TB (translation buffer) invalidate
whami	Who am I
wrasn	Write ASN
wrent	Write system entry address
wrfen	Write floating-point enable
wripir	Write interprocessor interrupt request
wrkgp	Write kernel global pointer
wrmces	Write machine check error summary register
wrperfmon	Performance monitoring function
wrsysptb	Write system page table base
wrusp	Write user stack pointer
wrval	Write system value
wrvirbnd	Write virtual address boundary
wrvptptr	Write virtual page table pointer
wtint	Wait for interrupt

21.2.1 Cache Flush

Format:

cflush

!PALcode format

Operation:

```
! a0 contains the page frame number (PFN)
!      of the page to be flushed
```

```
IF PS<mode> EQ 1 THEN
    {Initiate opDec fault}

    {Flush page out of cache(s)}
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

cflush

Cache flush

Description:

The cflush instruction may be used to flush an entire physical page specified by the PFN in a0 from any data caches associated with the current processor. All processors must implement this instruction.

On processors that implement a backup power option that maintains only the contents of memory if a powerfail occurs, this instruction is used by the powerfail interrupt handler to force data written by the handler to the battery backed-up main memory. After a cflush, the first subsequent load (on the same processor) to an arbitrary address in the target page is either fetched from physical memory or from the data cache of another processor.

In some multiprocessor systems, cflush is not sufficient to ensure that the data are actually written to memory and not exchanged between processor caches. Additional platform-specific cooperation between the powerfail interrupt handlers executing on each processor may be required.

On systems that implement other backup power options (including none), cflush may return without affecting the data cache contents.

To order cflush properly with respect to preceding writes, an MB instruction is needed before the cflush; to order cflush properly with respect to subsequent reads, an MB instruction is needed after the cflush.

21.2.2 Console Service

Format:

cserve

!PALcode format

Operation:

```
! implementation specific

if PS<mode> EQ 1 then
{initiate opDec fault}

else
{implementation-dependent action}
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

cserve Console service

Description:

This instruction is specific to each PALcode and console implementation and is not intended for operating system use.

21.2.3 Read Machine Check Error Summary

Format:

rdmces

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← MCES
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

rdmces Read machine check error summary

Description:

The read machine check error summary (rdmces) instruction returns the MCES (machine check error summary) register in v0. On return from the rdmces instruction, registers t0 and t8...t11 are UNPREDICTABLE.

21.2.4 Read Processor Status

Format:

rdps

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← PS
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

rdps Read processor status

Description:

The read processor status (rdps) instruction returns the PS in v0. On return from the rdps instruction, registers t0 and t8...t11 are UNPREDICTABLE.

21.2.5 Read User Stack Pointer

Format:

rdusp

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← USP
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

rdusp Read user stack pointer

Description:

The read user stack pointer (rdusp) instruction returns the user stack pointer in v0. The user stack pointer is written by the wrusp instruction, described in Section 21.2.22. On return from the rdusp instruction, registers t0 and t8...t11 are UNPREDICTABLE.

21.2.6 Read System Value

Format:

rdval

!PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← sysvalue
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

rdval Read system value

Description:

The read system value (rdval) instruction returns the sysvalue in v0, allowing access to a 64-bit per-processor value for use by the operating system. On return from the rdval instruction, registers t0 and t8...t11 are UNPREDICTABLE.

21.2.7 Return from System Call

Format:

retsys

! PALcode format

Operation:

```
if {PS<mode> EQ 1} then
    {Initiate opDec fault}
endif
tmp ← (SP+08)
GP ← (SP+16)
KSP ← SP + {6*8}
SP ← USP
intr_flag = 0           ! Clear the interrupt flag
lock_flag = 0         ! Clear the load lock flag
PS ← 8                ! Mode=user
PC ← tmp
```

Exceptions:

Opcode reserved to Compaq
Kernel stack not valid (halt)

Instruction Mnemonics:

retsys Return from system call

Description:

The return from system call (retsys) instruction pops the return address and the user mode global pointer from the kernel stack. It then saves the kernel stack pointer, sets the mode to user, sets the IPL to zero, and enters the user mode code at the address popped off the stack. On return from the retsys instruction, registers t0 and t8...t11 are UNPREDICTABLE.

21.2.8 Return from Trap, Fault or Interrupt

Format:

rti

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
tempps ← (SP+0)
temppc ← (SP+8)
GP ← (SP+16)
a0 ← (SP+24)
a1 ← (SP+32)
a2 ← (SP+40)
SP ← SP + {6 * 8}
if { tempps<3> EQ 1} then
    KSP ← SP          ! New mode is user
    SP ← USP
    tempps ← 8
endif
intr_flag = 0        ! Clear the interrupt flag
lock_flag = 0        ! Clear the load lock flag
PS ← tempps<3:0>     ! Set new PS
PC ← temppc
```

Exceptions:

Opcode reserved to Compaq

Kernel stack not valid (halt)

Instruction Mnemonics:

rti Return from trap, fault, or interrupt

Description:

The return from fault, trap, or interrupt (rti) instruction pops registers (a0...a2, and GP), the PC, and the PS, from the kernel stack. If the new mode is user, the kernel stack is saved and the user stack is restored.

21.2.9 Swap Process Context

Format:

swpctx

! PALcode format

Operation:

```
if (PS<mode> EQ 1)
    {Initiate opDec fault}
endif
(PCBB) ← SP                ! Save current state
(PCBB+8) ← USP
tmp ← PCC
tmp1 ← tmp<31:0> + tmp<63:32>
(PCBB+24)<31:0> ← tmp1<31:0>
v0 ← PCBB                  ! Return old PCBB
PCBB ← a0                  ! Switch PCBB
SP ← (PCBB)                ! Restore new state
USP ← (PCBB+8)
oldPTBR ← PTBR
PTBR ← (PCBB+16)
tmp1 ← (PCBB+24)
PCC<63:32> ← {tmp1 - tmp}<31:0>
FEN ← (PCBB+40)
if {process unique register implemented} then
    (v0+32) ← unique
    unique ← (PCBB+32)
endif
if {ASN implemented}
    ASN ← tmp1<63:32>
else
    if (oldPTBR NE PTBR)
        {Invalidate all TB entries with ASM=0}
    endif
endif
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

swpctx Swap process context

Description:

The swap process context (swpctx) instruction saves the current process data in the current PCB. Then swpctx switches to the PCB passed in a0 and loads the new process context. The old PCBB is returned in v0.

The process context and the PCB are described in Chapter 12.

On return from the swpctx instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

21.2.10 Swap IPL

Format:

swpipl

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← PS<IPL>
PS<IPL> ← a0<2:0>
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

swpipl Swap IPL

Description:

The swap IPL (swpipl) instruction returns the current value of the PS<IPL> bits in v0 and sets the IPL to the value passed in a0. On return from the swpipl instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

21.2.11 Swap PALcode Image

Format:

swppal

!PALcode format

Operation:

```
! a0 contains the new PALcode identifier
! a1:a5 contain implementation-specific entry parameters
! v0 receives the following status:
!     0 success (PALcode was switched)
!     1 unknown PALcode variant
!     2 known PALcode variant, but PALcode not loaded

if (PS<mode> EQ 1) then
    (Initiate opDec fault)

else
    if {a0 < 256} then
        begin
            if {a0 invalid} then
                v0 ← 1
                {return}
            else if {PALcode not loaded} then
                v0 ← 2
                {return}
            else
                tmp1 ← {PALcode base}
        end
    else
        tmp1 = a0
        {flush instruction cache}
        {invalidate all translation buffers}
        {perform additional PALcode variant-specific initialization}
        {transfer control to PALcode entry at physical address in tmp1}
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

swppal

Swap PALcode image

Description:

The swap Palcode image (swppal) instruction causes the current (active) PALcode to be replaced by the specified new PALcode image. The swppal instruction is intended for use by operating systems only during bootstraps and by consoles during transitions to console I/O mode.

The PALcode descriptor contained in a0 is interpreted as either a PALcode variant or the base physical address of the new PALcode image. If a variant, the PALcode image must have been loaded previously. No PALcode loading occurs as a result of this instruction.

After successful PALcode switching, the register contents are determined by the parameters passed in a1...a5 or are UNPREDICTABLE. A common parameter is the address of a new PCB. In this case, the stack pointer register and PTBR are determined by the contents of that PCB; the contents of other registers such as a0...a5 may be UNPREDICTABLE.

See Section 27.3.2, for information on using this instruction.

21.2.12 TB Invalidate

Format:

tbi ! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
case a0 begin
    1: ! tbisi
        {Invalidate ITB entry for va=a11}
        break;
    2: ! tbi sd
        {Invalidate DTB entry for va=a11}
        break;
    3: ! tbi s
        {Invalidate both ITB and DTB entry for va=a11}
        break;
    -1: ! tbi ap
        {Invalidate all TB entries with ASM=0}
        break;
    -2: ! tbi a
        {Flush all TBs}
        break;
    otherwise:
        break;
endcase
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

tbi TB (translation buffer) invalidate

Description:

The TB invalidate (tbi) instruction removes specified entries from the I and D translation buffers (TBs) when the mapping changes. The tbi instruction removes specific entry types based on a CASE selection of the value passed in register a0. On return from the tbi instruction, registers t0, t8...t11, a0, and a1 are UNPREDICTABLE. See Section 22.7 for information on translation buffers and Section 22.8 for information on address space numbers (ASNs), because ASNs can implicitly modify TB operations.

¹ Operation assumes no behavior modification from ASNs.

21.2.13 Who Am I

Format:

whami

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
v0 ← whami
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

whami

Who am I

Description:

The who am I (whami) instruction returns the processor number for the current processor in v0. The processor number is in the range 0 to the number of processors minus one (0...maxCPU-1) that can be configured in the system. On return from the whami instruction, registers t0 and t8...t11 are UNPREDICTABLE.

21.2.14 Write ASN

Format:

wrasn

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
```

```
ASN <- a0<31:0>
(PCBB+24)<63:32> <- a0<31:0>
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrasn Write ASN

Description:

The write ASN (wrasn) instruction writes a new ASN. It also writes the value for ASN to the PCB at offset (PCBB+24)<63:32>. On return from the wrasn instruction, registers t0, t8 ...t11, and a0 are UNPREDICTABLE.

21.2.15 Write System Entry Address

Format:

wrent

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
case a1 begin
    0: ! Write the EntInt:
        entInt ← a0
        break;
    1: ! Write the EntArith:
        entArith ← a0
        break;
    2: ! Write the EntMM:
        entMM ← a0
        break;
    3: ! Write the EntIF:
        entIF ← a0
        break;
    4: ! Write the EntUna:
        entUna ← a0
        break;
    5: ! Write the EntSys:
        entSys ← a0
        break;
    otherwise:
        break;
endcase;
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrent Write system entry address

Description:

The write system entry address (wrent) instruction determines the specific system entry point based on a CASE selection of the value passed in register a1. The wrent instruction then sets the virtual address of the specified system entry point to the value passed in a0.

For best performance, all the addresses should be kseg addresses. (See Section 22.1 for a definition of kseg addresses.) On return from the wrent instruction, registers t0, t8...t11, a0, and a1 are UNPREDICTABLE.

21.2.16 Write Floating-Point Enable

Format:

wrfen

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
FEN ← a0<0>
(PCBB+40)<0> ← a0 AND 1
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrfen Write floating-point enable

Description:

The write floating-point enable (wrfen) instruction writes bit zero of the value passed in a0 to the floating-point enable register. The wrfen instruction also writes the value for FEN to the PCB at offset (PCBB+40)<0>. On return from the wrfen instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

21.2.17 Write Interprocessor Interrupt Request

Format:

wripir

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
IPIR ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wripir Write interprocessor interrupt request

Description:

The write interprocessor interrupt request (wripir) instruction generates an interprocessor interrupt on the processor number passed in register a0. The interrupt request is recorded on the target processor and is initiated when the proper enabling conditions are present. On return from wripir, registers t0, t8...t11, and a0 are UNPREDICTABLE.

Programming Note:

The interrupt need not be initiated before the next instruction is executed on the requesting processor, even if the requesting processor is also the target processor for the request.

21.2.18 Write Kernel Global Pointer

Format:

wrkgp

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
KGP ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrkgp Write kernal global pointer

Description:

The write kernel global pointer (wrkgp) instruction writes the value passed in a0 to the kernel global pointer (KGP) internal register. The KGP is used to load the GP on exceptions. On return from the wrkgp instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

21.2.19 Write Machine Check Error Summary

Format:

wrmces

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
if (a0<0> EQ 1) then MCES<0> ← 0
if (a0<1> EQ 1) then MCES<1> ← 0
if (a0<2> EQ 1) then MCES<2> ← 0
MCES<3> ← a0<3>
MCES<4> ← a0<4>
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrmces Write machine check error summary

Description:

The write machine check error summary (wrmces) instruction 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 enabled bit in the MCES register. On return from the wrmces instruction, registers t0, t8...t11 are UNPREDICTABLE.

21.2.20 Performance Monitoring Function

Format:

wrperfmon

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
! a0 contains implementation specific input values
! a1 contains implementation specific output values
! v0 may return implementation specific values
! Operations and actions taken are implementation specific
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrperfmon

Performance monitoring

Description:

The performance monitoring instruction (wrperfmon) alerts any performance monitoring software/hardware in the system to monitor the performance of this process. The wrperfmon function arguments and actions are platform and chip dependent, and when defined for an implementation, are described in Appendix E.

Registers a0 and a1 contain implementation-specific input values. Implementation-specific values may be returned in register v0. On return from the wrperfmon instruction, registers a0, a1, t0, and t8...t11 are UNPREDICTABLE.

21.2.21 Write System Page Table Base

Format:

wrsysptb

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
```

```
SYSPTBR ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrsysptb Write system page table base

Description:

The write system page table base (wrsysptb) instruction writes the System Page Table Physical Base (SYSPTBR) register. It contains the page frame number (pfn) of the highest level page table to be used for system-wide addresses equal to or above the value of the Virtual Address Boundary Register. The System Page Table and Virtual Address Boundary base registers are described in Section 22.6.

On return from the wrsysptb instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

Note that this register is not context switched.

21.2.22 Write User Stack Pointer

Format:

wrusp

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
USP ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrusp Write user stack pointer

Description:

The write user stack pointer (wrusp) instruction writes the value passed in a0 to the user stack pointer. On return from the wrusp instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

21.2.23 Write System Value

Format:

wrval

!PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
sysvalue ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrval Write system value

Description:

The write system value (wrval) instruction writes the value passed in a0 to a 64-bit system value register. The combination of wrval with the rdval instruction, described in Section 21.2.6, allows access by the operating system to a 64-bit per-processor value. On return from the wrval instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

21.2.24 Write Virtual Address Boundary

Format:

wrvirbnd

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
```

VIRBND ← a0

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrvirbnd

Write virtual address boundary

Description:

The write virtual address boundary (wrvirbnd) instruction writes the virtual address boundary register (VIRBND), used to determine which page table physical base register is used. The System Page Table and Virtual Address Boundary base registers are described in Section 22.6. UNPREDICTABLE operations result if the address is not 64-bit aligned.

On return from the wrvirbnd instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

At processor initialization, VIRBND is initialized to a value of -1, which results in all translations using PTBR. The value in SYSPTBR is thus effectively ignored.

21.2.25 Write Virtual Page Table Pointer

Format:

wrvptptr

! PALcode format

Operation:

```
if (PS<mode> EQ 1) then
    {Initiate opDec fault}
endif
VPTPTR ← a0
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wrvptptr Write virtual page table pointer

Description:

The write virtual page table pointer (wrvptptr) instruction writes the pointer passed in a0 to the virtual page table pointer register (VPTPTR). The VPTPTR is described in Section 22.6.2. On return from the wrvptptr instruction, registers t0, t8...t11, and a0 are UNPREDICTABLE.

21.2.26 Wait for Interrupt

Format:

wtint ! PALcode format

Operation:

```
! a0 contains the maximum number of interval clock ticks to skip
! v0  receives the number of interval clock ticks actually skipped

IF (implemented)
BEGIN
  IF {Implementation supports skipping multiple
      clock interrupts} THEN
    {Ticks_to_skip ← a0}

    {Wait no longer than any non-clock interrupt or the first clock
     interrupt after ticks_to_skip ticks have been skipped}

  IF {Implementation supports skipping multiple
      clock interrupts} THEN
    v0 ← number of interval clock ticks actually skipped
  ELSE
    v0 ← 0
END
ELSE
  v0 ← 0
{return}
```

Exceptions:

Opcode reserved to Compaq

Instruction Mnemonics:

wtint Wait for interrupt

Description:

The wait for interrupt instruction (wtint) 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
- The first clock tick after a specified number of clock ticks has been skipped

The wtint instruction returns in v0 the number of clock ticks that are skipped. The number returned in v0 is zero on hardware platforms that implement this instruction, but where it is not possible to skip clock ticks.

The operating system can specify a full 64-bit integer value in `a0` as the maximum number of interval clock ticks to skip. A value of zero in `a0` causes no clock ticks to be skipped.

Note the following if specifying in `a0` the maximum number of interval clock ticks to skip:

- Adherence to a specified value in `a0` is at the discretion of the PALcode; the PALcode may complete execution of `wtint` and proceed to the next instruction at any time up to the specified maximum, even if no interrupt or interval-clock tick has occurred. That is, `wtint` may return before all requested clock ticks are skipped.
- The PALcode must complete execution of `wtint` if an interrupt occurs or if an interval-clock tick occurs after the requested number of interval-clock ticks has been skipped.

In a multiprocessor environment, only the issuing processor is affected by an issued `wtint` instruction.

The counter, `PCC`, may increment at a lower rate or may stop entirely during `wtint` execution. This side effect is implementation dependent.

Memory Management (II–C)

22.1 Virtual Address Spaces

A virtual address is a 64-bit unsigned integer that specifies a byte location within the virtual address space. Implementations subset the supported address space to one of several sizes, as a function of page size and page table depth. The minimal supported virtual address size is 43 bits. If an implementation supports less than 64-bit virtual addresses, it must check that all the $VA\langle 63:vaSize\rangle$ bits are equal to $VA\langle vaSize-1\rangle$. This gives two disjoint ranges for valid virtual addresses. For example, for a 43-bit virtual address space, valid virtual address ranges are $0\dots 3FFFFFFF_{16}$ and $FFFFFC0000000000_{16}\dots FFFFFFFF_{16}$. Access to virtual addresses outside an implementation's valid virtual address range cause an access-violation fault¹.

The virtual address space is divided into three segments: seg0, seg1, and kseg.

The two bits, $va\langle vaSize-1:vaSize-2\rangle$, select a segment as shown in Table 22–1.

Table 22–1 Virtual Address Space Segments

$VA\langle vaSize-1:vaSize-2\rangle$	Name	Mapping	Access Control
00	seg0	Mapped via 3 levels of PTEs	Programmed in PTE
01	seg0	Mapped via 2 levels of PTEs	Programmed in PTE
10	kseg	$PA \leftarrow SEXT(VA\langle vaSize-3:0\rangle)$	Kernel Read/Write
11	seg1	Mapped via the TB	Programmed in PTE

For kseg, the relocation, sharing, and protection are fixed. The base of kseg is located at $LEFT_SHIFT(FFFFFC0000000000_{16}, (vaSize-43))$.

For seg0 and seg1, the virtual address space is broken into pages, which are the units of relocation, sharing, and protection. The page size ranges from 8K bytes to 64K bytes. Therefore, system software should allocate regions with differing protection on 64K-byte virtual address boundaries to ensure image compatibility across all Alpha implementations.

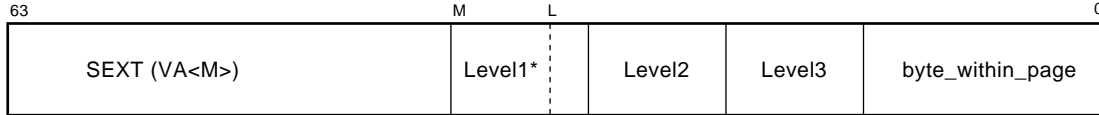
Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. The operating system controls the virtual-to-physical address mapping tables and saves the inactive (but used) parts of the virtual address space on external storage media.

¹ The highest physical address that can be addressed by kseg in 43-bit addressing mode can be extended, under certain circumstances, by an optional 48-bit/43-bit virtual addressing mode, described in Section E.2.1.

22.1.1 Segment Seg0 and Seg1 Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. A seg0 or seg1 virtual address consists of three level-number fields and a byte_within_page field, as shown in Figure 22–1 .

Figure 22–1 Virtual Address Format



* Level1 <M:L+1> contains SEXT(VA<L>), where L is the highest numbered implemented VA bit.

The byte_within_page field can be either 13, 14, 15, or 16 bits depending on a particular implementation. Thus, the allowable page sizes are 8K bytes, 16K bytes, 32K bytes, and 64K bytes. The low-order bit in each level-number field is 0 and each field is 0...n bits, where for example, n is 9 for an 8K page size.

An implementation may support a smaller virtual address space than the page size allows by including only a subset of low-order bits in Level1. The smaller virtual address space must be at least 43 bits and must be large enough that at least two bits of Level1 are implemented.

The level-number fields are a function of the page size; all page table entries at any given level do not exceed one page. The PFN field in the PTE is always 32 bits wide. Thus, as the page size grows, the virtual and physical address size also grows.

Table 22–2 shows the virtual address options and physical address size (in bits) calculations. The physical address (bits) column is the maximum physical address allowed by the smaller of the kseg size or available physical address bits for a given page size. The available physical address bits is calculated by combining the number of bits in the PFN (always 32) with the number of bits in the byte_within_page field. The kseg segment size is calculated from the virtual address size minus 2.

Table 22–2 Virtual Address Options

Page Size (bytes)	Byte_within_page (bits)	Level Size (bits)	Virtual Address (bits)	Maximum Physical Address (bits)	Physical Address Limited by
8K	13	10	43	41	kseg
16K	14	11	43–47 ¹	45	kseg
32K	15	12	43–51 ¹	47	seg0/seg1
64K	16	13	44–55 ¹	48	seg0/seg1

¹ Level1 page table might be partially utilized for this page size.

22.1.2 Kseg Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. A kseg virtual address consists of segment select field with a value of 10_2 and a physical address field. The segment select field is the two bits $va\langle vaSize-1:vaSize-2 \rangle$. The physical address field is $va\langle vaSize-3:0 \rangle$.

Figure 22–2 Kseg Virtual Address Format



22.2 Physical Address Space

Physical addresses are at most $vaSize-2$ bits. This allows all of physical memory to be accessed via kseg. A processor may choose to implement a smaller physical address space by not implementing some number of high-order bits.

The two most significant implemented physical address bits delineate the four regions in the physical address space. Implementations use these bits as appropriate for their systems. For example, in a workstation with a 30-bit physical address space, bit $\langle 29 \rangle$ might select between memory and non-memory-like regions, and bit $\langle 28 \rangle$ could enable or disable cacheing (see Section 5.2.4).

22.3 Memory Management Control

Memory management is always enabled. Implementations must provide an environment for PALcode to service exceptions and to initialize and boot the processor. For example PALcode might run with I-stream mapping disabled.

22.4 Page Table Entries

The processor uses a quadword page table entry (PTE) to translate $seg0$ and $seg1$ virtual addresses to physical addresses. A PTE contains hardware and software control information and the physical page frame number (PFN). A PTE is a quadword with fields as shown in Figure 22–3 and described in Table 22–3.

Figure 22–3 Page Table Entry (PTE)

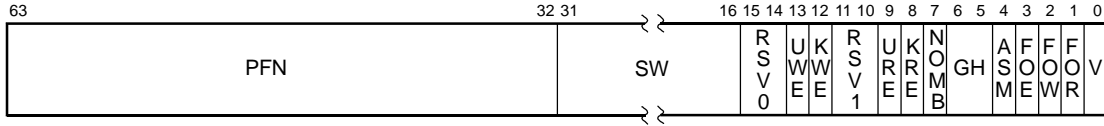


Table 22–3 Page Table Entry (PTE) Bit Summary

Bits	Name	Meaning
63–32	PFN	Page frame number. The PFN field always points to a page boundary. If V is set, the PFN is concatenated with the byte_within_page bits of the virtual address to obtain the physical address.
31–16	SW	Reserved for software.
15–14	RSV0	Reserved for hardware; SBZ.
13	UWE	User write enable. Enables writes from user mode. If this bit is 0 and a store is attempted while in user mode, an access-violation fault occurs. This bit is valid even when V=0.
Note:		
If a write enable bit is set and the corresponding read enable bit is not, the operation of the processor is UNDEFINED.		
12	KWE	Kernel write enable. Enables writes from kernel mode. If this bit is 0 and a store is attempted while in kernel mode, an access-violation fault occurs. This bit is valid even when V=0.
11–10	RSV1	Reserved for hardware; SBZ.
9	URE	User read enable. Enables reads from user mode. If this bit is 0 and a load or instruction fetch is attempted while in user mode, an Access Violation occurs. This bit is valid even when V=0.
8	KRE	Kernel read enable. Enables reads from kernel mode. If this bit is 0 and a load or instruction fetch is attempted while in kernel mode, an access-violation fault occurs. This bit is valid even when V=0.
7	NOMB	Translation buffer miss memory barrier. When set, the requirement described in Section 5.6.4.3 is lifted for ensuring that all processors using a newly valid PTE also see any new contents of the related page. This allows the TB-miss code to avoid potentially expensive global synchronization. Software is expected to set this bit on PTEs when it is known that the page contents are already visible to all processors.

Table 22–3 Page Table Entry (PTE) Bit Summary (Continued)

Bits	Name	Meaning																																		
6–5	GH	Granularity hint (GH). Software may set these bits as follows to supply a hint to translation buffer implementations that a block of pages can be treated as a single larger page:																																		
		<table border="1"> <thead> <tr> <th rowspan="2">PTE<6:5></th> <th colspan="4">Page Size Before GH:</th> </tr> <tr> <th>8KB</th> <th>16KB</th> <th>32KB</th> <th>64KB</th> </tr> </thead> <tbody> <tr> <td></td> <td colspan="4">Resulting Page Size:</td> </tr> <tr> <td>00</td> <td>8KB</td> <td>16KB</td> <td>32KB</td> <td>64KB</td> </tr> <tr> <td>01</td> <td>64B</td> <td>128KB</td> <td>256KB</td> <td>2MB</td> </tr> <tr> <td>10</td> <td>512KB</td> <td>1MB</td> <td>2MB</td> <td>64MB</td> </tr> <tr> <td>11</td> <td>4MB</td> <td>8MB</td> <td>16MB</td> <td>512MB</td> </tr> </tbody> </table>	PTE<6:5>	Page Size Before GH:				8KB	16KB	32KB	64KB		Resulting Page Size:				00	8KB	16KB	32KB	64KB	01	64B	128KB	256KB	2MB	10	512KB	1MB	2MB	64MB	11	4MB	8MB	16MB	512MB
PTE<6:5>	Page Size Before GH:																																			
	8KB	16KB	32KB	64KB																																
	Resulting Page Size:																																			
00	8KB	16KB	32KB	64KB																																
01	64B	128KB	256KB	2MB																																
10	512KB	1MB	2MB	64MB																																
11	4MB	8MB	16MB	512MB																																

Notes:

1. The block is a group of physically contiguous pages that are naturally aligned both virtually and physically. Within the block, the PFN field in each PTE must map the correct physical page for the virtual page to which the PTE corresponds.
2. Within the block, all PTEs have the same values for bits <15:0>, that is, protection, fault, granularity, and valid bits.

Hardware may use this hint to map the entire block with a single TB entry.

It is UNPREDICTABLE which PTE values within the block are used if the granularity bits are set inconsistently.

Programming Note:

A granularity hint might be appropriate for a large memory structure such as a frame buffer or nonpaged pool that, in fact, is mapped into contiguous virtual pages with identical protection, fault, and valid bits.

4	ASM	Address space match. When set, this PTE matches all address space numbers. For a given VA, ASM must be set consistently in all processes; otherwise, the address mapping is UNPREDICTABLE.
3	FOE	Fault on execute. When set, a Fault on Execute exception occurs on an attempt to execute any location in the page.

Table 22–3 Page Table Entry (PTE) Bit Summary (Continued)

Bits	Name	Meaning
2	FOW	Fault on write. When set, a Fault on Write exception occurs on an attempt to write any location in the page.
1	FOR	Fault on read. When set, a Fault on Read exception occurs on an attempt to read any location in the page.
0	V	Valid. Indicates the validity of the PFN field. When V is set, the PFN field is valid for use by hardware. When V is clear, the PFN field is reserved for use by software. The V bit does not affect the validity of PTE<15:1> bits.

22.4.1 Changes to Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, the operating system may set or clear the V bit, change the PFN field as pages are moved to and from external storage media, or modify the software bits. The processor hardware never changes PTEs.

Software must guarantee that each PTE is always internally consistent. Changing a PTE one field at a time can cause incorrect system operation, such as setting PTE<V> with one instruction before establishing PTE<PFN> with another. Execution of an interrupt service routine between the two instructions could use an address that would map using the inconsistent PTE. Software can solve this problem by building a complete new PTE in a register and then moving the new PTE to the page table by using an STQ instruction.

Multiprocessing complicates the problem. Another processor could be reading (or even changing) the same PTE that the first processor is changing. Such concurrent access must produce consistent results. Software must use some form of software synchronization to modify PTEs that are already valid. Whenever a processor modifies a valid PTE, it is possible that other processors in a multiprocessor system may have old copies of that PTE in their translation buffer. When software changes a PTE, each processor may use either the old or the new PTE until software performs a TB invalidate on that processor (after which, the processor may use only the new PTE). An example of a case where either the old or new PTE could usefully be used is when the NOMB bit is transitioned from zero to one. Hardware must ensure that aligned quadword reads and writes are atomic operations. Hardware must not cache invalid PTEs (PTEs with the V bit equal to 0) in translation buffers. See Section 22.7 for more information.

22.5 Memory Protection

Memory protection is the function of validating whether a particular type of access is allowed to a specific page from a particular access mode. Access to each page is controlled by a protection code that specifies, for each access mode, whether read or write references are allowed. The processor uses the following to determine whether an intended access is allowed:

- The virtual address, which is used to either select kseg mapping or provide the index into the page tables

- The intended access type (read or write)
- The current access mode base on processor mode

For protection checks, the intended access is read for data loads and instruction fetches, and write for data stores.

22.5.1 Processor Access Modes

There are two processor modes, user and kernel. The access mode of a running process is stored in the processor status mode bit (PS<mode>).

22.5.2 Protection Code

Every page in the virtual address space is protected according to its use. A program may be prevented from reading or writing portions of its address space. A protection code associated with each page describes the accessibility of the page for each processor mode.

For seg0 and seg1, the code allows a choice of read or write protection for each processor mode. For each mode, access can be read/write, read-only, or no-access. Read and write accessibility and the protection for each mode are specified independently.

For kseg, the protection code is kernel read/write, user no-access.

22.5.3 Access-Violation Faults

An access-violation memory-management fault occurs if an illegal access is attempted, as determined by the current processor mode and the page's protection.

22.6 Address Translation for Seg0 and Seg1

The page tables can be accessed from physical memory, or (to reduce overhead) can be mapped to a linear region of the virtual address space.

Additionally, an optional reduced page table (RPT) mode is defined, which allows more efficient mapping of very large blocks of memory.

The following sections describe the access methods.

22.6.1 Physical Access for Seg0 and Seg1 PTEs

In systems with Virtual Address Boundary and System Page Table Base registers, the virtual-address is compared against the Virtual Address Boundary register. Lower addresses use the PTBR as a physical page table base; higher or equal addresses use the SYSPTBR register.

Seg0 and seg1 address translation can be performed by accessing entries in a multilevel page table structure. The page table base register (PTBR or SYSPTBR) contains the physical page frame number (PFN) of the highest-level (Level 1) page table.

Bits <Level1> of the virtual address are used to index into the Level 1 page table to obtain the physical PFN of the base of the next level (Level 2) page table. Bits <Level2> of the virtual address are used to index into the Level 2 page table to obtain the physical PFN of the base of the next level (Level 3) page table. Bits <Level3> of the virtual address are used to index the

Level 3 page table to obtain the physical PFN of the page being referenced. The PFN is concatenated with virtual address bits <byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table does not reside in a memory-like region, or does reside in nonexistent memory, the operation of the processor is UNDEFINED.

If all the first- and second-level PTEs are valid, the protection bits are ignored; the protection code in the Level 3 PTE is used to determine accessibility. If a higher-level PTE (numerically, any below Level 3) is invalid, an access-violation fault occurs if the PTE<KRE> equals zero. An access-violation fault on any higher-level PTE implies that all lower-level page tables mapped by that PTE do not exist.

The algorithm to generate a physical address from a seg0 or seg1 virtual address follows:

```

IF {SEXT(VA<vaSize-1>:0) neq VA} THEN
  { initiate access-violation fault}
IF (VIRBND in use) THEN
  IF (VA LTU VIRBND) THEN
    ptbr_value <- PTBR
  ELSE
    ptbr_value <- SYSPTBR
ELSE
  ptbr_value <- PTBR

! Read Physical
level1_pte ←( { ptbr_value * page_size} + { 8 * VA<level1} )

IF level1_pte<v> EQ 0 THEN
  IF level1_pte<KRE> eq 0 THEN
    { initiate access-violation fault}
  ELSE
    { initiate translation-not-valid fault}
! Read physical:
level2_pte " ({level1_pte<PFN> * page_size} + {8 * VA<level2>} )
IF level2_pte<v> EQ 0 THEN
  IF level2_pte<KRE> eq 0 THEN
    { initiate access-violation fault}
  ELSE
    { initiate translation-not-valid fault}
! Read physical:
level3_PTE ← ({level2_pte<PFN> * page_size} + {8 * VA<level3>} )

IF {{{level3_pte<UWE> eq 0}AND {write access} AND {ps<mode> EQ 1}} OR
  {{{level3_pte<URE> eq 0} AND {read access} AND {ps<mode> EQ 1}} OR
  {{{level3_pte<KWE> eq 0}AND {write access} AND {ps<mode> EQ 0}} OR
  {{{level3_pte<KRE> eq 0}AND {read access} AND {ps<mode> EQ 0}}}
THEN
  {initiate memory-management fault}
ELSE
  IF level3_pte<v> EQ 0 THEN
    {initiate memory-management fault}

```

```

IF { level3_pte<FOW> eq 1 } AND {write access} THEN
    {initiate memory-management fault}
IF { level3_pte<FOR> eq 1 } AND {read access} THEN
    {initiate memory-management fault}
IF { level3_pte<FOE> eq 1 } AND {execute access} THEN
    {initiate memory-management fault}

```

```
Physical_address ← {level3_pte<PFN> * page_size} OR VA<byte_within_page>
```

22.6.2 Virtual Access for Seg0 or Seg1 PTEs

The page tables can be mapped into a linear region of the virtual address space, reducing the overhead for seg0 and seg1 PTE accesses. If SYSPTBR and VIRBND are implemented, care must be taken to ensure that the Level 3 page tables defined by both PTBR and SYSPTBR are mapped at the same virtual address. This is required so a single VPTPTR can be used regardless of which base register is determined to be used based on the value in VIRBND. (The physical PTE fetch defined in Section 22.6.1 enter the proper mappings into the TB.) The SYSPTBR and VIRBND registers are written by the wrsysptb and wrvirbnd PALcode instructions, described in Sections 21.2.21 and 21.2.24, respectively.

The mapping must be created exactly as follows because PALcode implementations may depend on details of the mapping.

1. Select a $2^{(3*\lg(\text{pageSize}/8)+3)}$ byte-aligned region (an address with $3*\lg(\text{pageSize}/8)+3$ low-order zeros) in the seg0 or seg1 address space.
2. Create a PTE in each of the page tables defined by PTBR and SYSPTBR (if implemented) to map the page tables as follows.

```

PTE = 0                ! Initialize all fields to zero
! Set the PFN to the Level 1 pagetable:
PTE<63:32> = PFN of Level 1 pagetable
PTE<8> = 1             ! Set the kernel read enable bit
PTE<0> = 1             ! Set the valid bit

```

3. Set the page table entry that corresponds to the VPTPTR to the created Level 1 PTE.
4. Set all Level 1 and Level 2 valid PTEs to allow kernel read access. With this setup in place, the algorithm to fetch a seg0 or seg1 PTE is as follows, where pS represents pageSize :

```

tmp ← LEFT_SHIFT (va, {64 - {{lg(pS)* 4} - 9}})
tmp ← RIGHT_SHIFT (tmp, {64 - {{lg(pS)* 4} - 9} + lg(pS)-3})
tmp ← VPTB OR tmp
tmp<2:0> ← 0
level3_PTE ← (tmp)          ! Load PTE using its virtual address

```

5. Set the virtual page table pointer (VPTPTR) with a write virtual page table pointer instruction (wrvp ptr) to the selected value.

The virtual access method is used by PALcode for most TB fills.

Implementation Note:

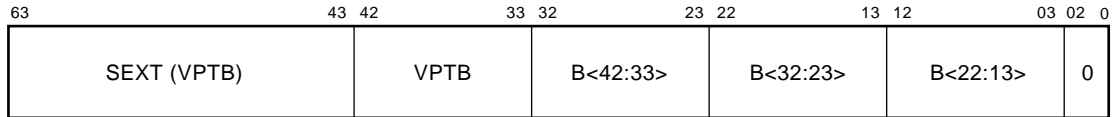
Assume the following:

- A system with a 52-bit virtual address size.

- VPTB is the index of the Level 1 PTE, which is self-referencing.
- The virtual address is in seg0 or seg1.

For a virtual address B, the address to virtually access the Level 3 PTE is as follows. The double-miss TB fill flow is a three-level flow.

Figure 22–4 Three-Level Page Table Mapping



22.6.3 Reduced Page Table (RPT) Mode

The reduced page table (RPT) mode is an optional extension of 64KB page size mode. A portion of the address space is mapped by one fewer page table levels, allowing each of the entries in the lowest-level page table to map a 512MB page. In implementations that support granularity hints in hardware, applications can use these hints to make more efficient use of the translation buffer. Applications that can use the 512MB granularity hint in 64KB page size mode can use RPT mode for additional benefits.

With the 512MB granularity hint but without RPT, every entry in the Level3 page table maps the same 512MB page. With RPT, that Level3 page table is eliminated entirely, and the Level2 PTE that would normally point to that Level3 page table is used to directly map the 512MB page.

Therefore, in an RPT region, there is elimination of redundant page table pages and compression of page table space. The compressed PTEs are more likely to fit in hardware caches. If there is locality of reference, a new PTE that is needed to satisfy a mapping is more likely to be present in the cache. Additionally, a single TB entry that maps the VA of the lowest-level page table now allows access to PTEs mapping 4TB, rather than 512MB, of memory.

In order to use RPT mode, the feature must be available and enabled in the implementation, and:

- Use the 64KB page size.
- Every L2 PTE in the reduced page table region must have $PTE\langle GH \rangle = 11_2$, that is, a 512MB page size.
- The PFN field of the PTE must refer to a 512 MB aligned page.
- The RPT region is selected by using VAs such that $VA\langle vaSize-1:vaSize-2 \rangle = 01_2$.

22.6.3.1 Physical Access for Page Table Entries in Reduced Page Table Mode

Physical address translation is performed by accessing entries in a two-level page table structure. The Page Table Base Register (PTBR) contains the physical Page Frame Number (PFN) of the highest-level (Level1) page table.

In systems that implement the Virtual Address Boundary register (VIRBND), the System Page Table Base Register (SYSPTBR) contains the PFN of an alternate highest-level page table. In such systems, the virtual address to be translated is compared against the address stored in VIRBND. Translations of Level2 addresses begin with the PFN in PTBR as the highest-level

page table. Translations of Level1 addresses use the PFN in SYSPTBR as the highest-level page table. The VIRBND and SYSPTBR registers are described in Sections 13.3.24 and 13.3.18, respectively.

Level1 is the highest-level page table. Bits <Level1> of the virtual address are used to index into the Level1 page table to obtain the physical PFN of the base of the next level (Level2) page table. Bits <Level2> of the virtual address are used to index into the Level2 page table to obtain the physical PFN of the page being referenced. The PFN is concatenated with virtual address bits <byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table resides in I/O space, or in nonexistent memory, the operation of the processor is UNDEFINED.

If the Level1 PTE is valid, the protection bits are ignored; the protection code in the Level2 PTE is used to determine accessibility. If a Level1 PTE is invalid, an access-violation fault occurs if the PTE<KRE> equals zero. An access-violation fault on any Level1 PTE implies that all Level2 page tables mapped by that PTE do not exist.

The algorithm to generate a physical address from a virtual address follows:

```

IF {SEXT(VA<vaSize-1>:0) neq VA} THEN
  { initiate access-violation fault}
IF (VIRBND in use) THEN
  IF (VA LTU VIRBND) THEN
    ptbr_value <- PTBR
  ELSE
    ptbr_value <- SYSPTBR
ELSE
  ptbr_value <- PTBR

! Read Physical
level1_pte ←( { ptbr_value * page_size} + { 8 * VA<level1} )

IF level1_pte<v> EQ 0 THEN
  IF level1_pte<KRE> eq 0 THEN
    { initiate access-violation fault}
  ELSE
    { initiate translation-not-valid fault}
! Read physical:
level2_pte " ({level1_pte<PFN> * page_size} + {8 * VA<level2>} )

IF {{{level2_pte<UWE> eq 0}AND {write access} AND {ps<mode> EQ 1}} OR
  {{{level2_pte<URE> eq 0} AND {read access} AND {ps<mode> EQ 1}} OR
  {{{level2_pte<KWE> eq 0}AND {write access} AND {ps<mode> EQ 0}} OR
  {{{level2_pte<KRE> eq 0}AND {read access} AND {ps<mode> EQ 0}}}
  THEN
    {initiate memory-management fault}
  ELSE
    IF level2_pte<v> EQ 0 THEN
      {initiate memory-management fault}

IF { level2_pte<FOW> eq 1} AND {write access} THEN
  {initiate memory-management fault}
IF { level2_pte<FOR> eq 1} AND {read access} THEN
  {initiate memory-management fault}

```

```

IF { level2_pte<FOE> eq 1 } AND {execute access} THEN
    {initiate memory-management fault}
Physical_Address ← {level2_pte<PFN> * page_size} OR VA<byte_within_RPT_page1>

```

22.6.3.2 Virtual Access for Page Table Entries in Reduced Page Table Mode

To reduce overhead associated with the address translation in a multilevel page table structure, the page tables are mapped into a linear region of the virtual address space. The virtual address of the base of the page table structure is set on a system-wide basis and is contained in the VPTB IPR.

When a native mode DTB or ITB miss occurs, it is desirable that the TBMISSE flow attempt to load the lowest-level PTE by using a single virtual load instruction without regard to whether the missing VA is mapped by two levels (RPT) or three levels of page table. (See Section E.2.2 for the 21364 implementation.)

22.7 Translation Buffer

In order to save actual memory references when repeatedly referencing the same pages, hardware implementations include a translation buffer to remember successful virtual address translations and page states.

When the process context is changed, a new value is loaded into the address space number (ASN) internal processor register with a swap process context (swpctx) instruction. This causes address translations for pages with PTE<ASM> clear to be invalidated on a processor that does not implement address space numbers.

Additionally, when the software changes any part (except the software field) of a valid PTE, it must also execute a tbi instruction. The entire translation buffer can be invalidated by tbia, and all ASM=0 entries can be invalidated by tbiap. The translation buffer must not store invalid PTEs. Therefore, the software is not required to invalidate translation buffer entries when making changes for PTEs that are already invalid. Changes to PTE<NOMB> are also an exception to this requirement. This bit only has an effect when a PTE is loaded into the translation buffer. Thus, there is no need to invalidate the TB when the bit changes.

After software changes a valid first-, or second-level PTE, software must flush the translation for the corresponding page in the virtual page table. Then software must flush the translations of all valid pages mapped by that page. In the case of a change to a first-level PTE, this action must be taken through a second iteration.

22.8 Address Space Numbers

The Alpha architecture allows a processor to optionally implement address space numbers (process tags) to reduce the need for invalidation of cached address translations for process-specific addresses when a context switch occurs. The supported address space number (ASN) range is 0...MAX_ASN; MAX_ASN is provided in the HWRPB MAX_ASN field.

¹ byte_within_RPT_page contains those bits that would have been VA<Level3>, concatenated with the VA<byte_within_page> field for 64KB page table mode .

The address space number for the current process is loaded by software in the address space number (ASN) with a swpctx instruction. ASNs are processor specific and the hardware makes no attempt to maintain coherency across multiple processors. In a multiprocessor system, software is responsible for ensuring the consistency of TB entries for processes that might be rescheduled on different processors.

Systems that support ASNs should have MAX_ASN in the range 13...65535. The number of ASNs should be determined by the market a system is targeting.

Programming Note:

System software should not assume that the number of ASNs is a power of two. This allows hardware, for example, to use N TB tag bits to encode $(2^{**N})-3$ ASN values, one value for ASM=1 PTEs, and one for invalid.

There are several possible ways of using ASNs that result from several complications in a multiprocessor system. Consider the case where a process that executed on processor-1 is rescheduled on processor-2. If a page is deleted or its protection is changed, the TB in processor-1 has stale data.

- One solution is to send an interprocessor interrupt to all the processors on which this process could have run and cause them to invalidate the changed PTE. That results in significant overhead in a system with several processors.
- Another solution is to have software invalidate all TB entries for a process on a new processor before it can begin execution, if the process executed on another processor during its previous execution. This ensures the deletion of possibly stale TB entries on the new processor.
- A third solution is to assign a new ASN whenever a process is run on a processor that is not the same as the last processor on which it ran.

22.9 Memory-Management Faults

On a memory-management fault, the fault code (MMCSR) is passed in a1 to specify the type of fault encountered, as shown in Table 22-4.

Table 22-4: Memory-Management Fault Type Codes

Fault	MMCSR Value
Translation not valid	0
Access-violation	1
Fault on read	2
Fault on execute	3
Fault on write	4

- A translation-not-valid fault is taken when a read or write reference is attempted through an invalid PTE in a zero (if one exists), first, second, or third-level page table.

- An access-violation (ACV) fault is taken under the following circumstances:
 - An ACV fault is taken on a reference to a seg0 or seg1 address when the protection field of the third-level PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode.
 - An ACV fault is taken if the KRE bit is a zero in an invalid first-, or second-level PTE. An access-violation fault is generated for any access to a kseg address when the mode is user (PS<mode> EQ 1).
 - For reduced page table regions:
 - An ACV fault is taken when the protection field of the Level2 PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode.
 - An ACV fault is also taken if the KRE bit is zero in an invalid Level1 PTE.
- A fault-on-read (FOR) fault occurs when a read is attempted with PTE<FOR> set.
- A fault-on-execute (FOE) fault occurs when an instruction fetch is attempted with PTE<FOE> set.
- A fault-on-write (FOW) fault occurs when a write is attempted with PTE<FOW> set.

Process Structure (II–C)

23.1 Process Definition

A process is a single thread of execution. It is the basic entity that can be scheduled and is executed by the processor. A process consists of an address space and both software and hardware context. The hardware context of a process is defined by the following:

- Thirty integer registers (excludes R31 and SP)
- Thirty-one floating-point registers (excludes F31)
- The program counter (PC)
- The two per-process stack pointers (USP/KSP)
- The processor status (PS)
- The address space number (ASN)
- The charged process cycles
- The page table base register (PTBR)
- The process unique value (unique)
- The floating-point enable register (FEN)
- The performance monitoring enable bit (PME)

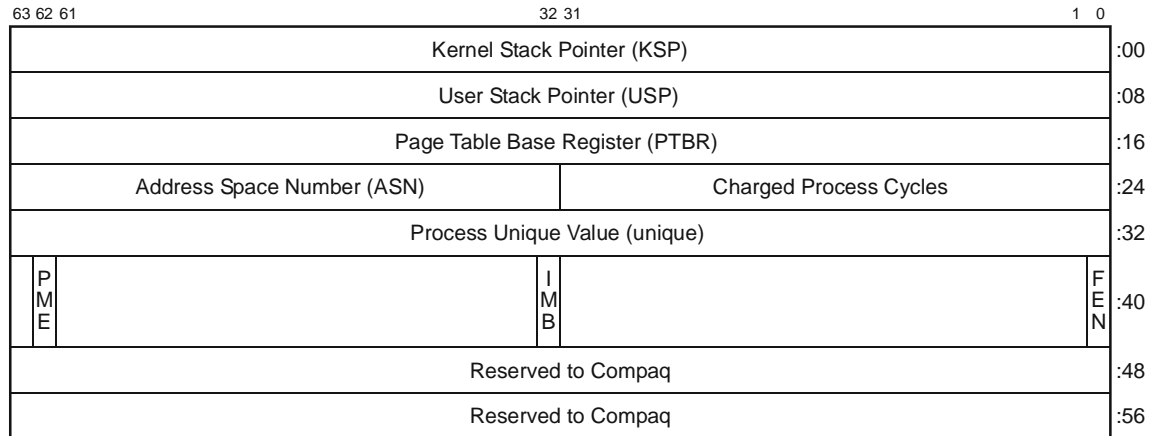
This information must be loaded if a process is to execute.

While a process is executing, some of its hardware context is being updated in the internal registers. When a process is not being executed, its hardware context is stored in memory in a software structure called the process control block (PCB). Saving the process context in the PCB and loading new values from another PCB for a new context is called context switching. Context switching occurs as one process after another is scheduled for execution.

23.2 Process Control Block (PCB)

As shown in Figure 23–1, the PCB holds the state of a process.

Figure 23–1 Process Control Block (PCB)



The contents of the PCB are loaded and saved by the swap process context (swpctx) instruction. The PCB must be quadword aligned and lie within a single page of physical memory. It should be 64-byte aligned for best performance.

The PCB for the current process is specified by the process control block base address register (PCBB); see Table 20–3.

The swap privileged context instruction (swpctx) saves the privileged context of the current process into the PCB specified by PCBB, loads a new value into PCBB, and then loads the privileged context of the new process into the appropriate hardware registers.

The new value loaded into PCBB, as well as the contents of the PCB, must satisfy certain constraints or an UNDEFINED operation results:

1. The physical address loaded into PCBB must be quadword aligned and describes eight contiguous quadwords that are in a memory-like region (see Section 5.2.4).
2. The value of PTBR must be the page frame number (PFN) of an existent page that is in a memory-like region.

It is the responsibility of the operating system to save and load the non-privileged part of the hardware context.

The swpctx instruction returns ownership of the current PCB to operating system software and passes ownership of the new PCB from the operating system to the processor. Any attempt to write a PCB while ownership resides with the processor has UNDEFINED results. If the PCB is read while ownership resides with the processor, it is UNPREDICTABLE whether the original or an updated value of a field is read. The processor is free to update a PCB field at any time. The decision as to whether or not a field is updated is made individually for each field.

The charged process cycles is the total number of PCC register counts that are charged to the process (modulo 2^{32}). When a process context is loaded by the swpctx instructions, the contents of the PCC count field (PCC_CNT) is subtracted from the contents of PCB[24]<31:0> and the result is written to the PCC offset field (PCC_OFF):

$PCC\langle 63:32 \rangle \leftarrow (PCB[24]\langle 31:0 \rangle - PCC\langle 31:0 \rangle)$

When a process context is saved by the swpctx instruction, the charged process cycles is computed by performing an unsigned add of $PCC\langle 63:32 \rangle$ and $PCC\langle 31:0 \rangle$. That value is written to $PCB[24]\langle 31:0 \rangle$.

Software Programming Note:

The following example returns in R0 the current PCC register count (modulo 2^{**32}) for a process. Notice the care taken not to cause an unwanted sign extension.

```
RPCC    R0                ; Read the processor 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
```

If ASNs are not implemented, the ASN field is not read or written by PALcode.

The process unique value is that value used in support of multithread implementations. The value is stored in the PCB when the process is not active. When the process is active, the value may be cached in hardware internal storage or kept in the PCB only.

The FEN bit reflects the setting of the FEN IPR.

The IMB bit records that an IMB was issued in user mode.

Setting the PME bit alerts any performance hardware or software in the system to monitor the performance of this process.

Kernel mode code must use the rdusp/wrusp instructions to access the USP. Kernel mode code can read the PTBR, the ASN, the FEN, and the PME for the current process from the PCB. The unique value can be accessed with the rdunique and wrunique instructions.

Exceptions and Interrupts (II–C)

24.1 Introduction

At certain times during the operation of a system, events within the system require the execution of software outside the explicit flow of control. When such an event occurs, an Alpha processor forces a change in control flow from that indicated by the current instruction stream. The notification process for such an event is either an exception or an interrupt.

24.1.1 Exceptions

Exceptions occur primarily in relation to the currently executing process. Exception service routines execute in response to exception conditions caused by software. All exception service routines execute in kernel mode on the kernel stack. Exception conditions consist of faults, arithmetic traps, and synchronous traps:

- A fault occurs during an instruction and leaves the registers and memory in a consistent state such that elimination of the fault condition and subsequent reexecution of the instruction gives correct results. Faults are not guaranteed to leave the machine in exactly the same state it was in immediately prior to the fault, but rather in a state such that the instruction can be correctly executed if the fault condition is removed. The PC saved in the exception stack frame is the address of the faulting instruction. An `rti` instruction to that PC reexecutes the faulting instruction.
- An arithmetic trap occurs at the completion of the operation that caused the exception. Since several instructions may be in various stages of execution at any point in time, it is possible for multiple arithmetic traps to occur simultaneously.

The PC that is saved in the exception frame on traps is that of the next instruction that would have been issued if the trapping conditions had not occurred. However, that PC is not necessarily the address of the instruction immediately following the instruction that encountered the trap condition, and the intervening instructions are collectively called the *trap shadow*. See Section 4.7.7.3 for information.

The intervening instructions may have changed operands or other state used by the instructions encountering the trap conditions. If such is the case, an `rti` instruction to that PC does not reexecute the trapping instructions, nor does it reexecute any intervening instructions; it simply continues execution from the point at which the trap was taken.

In general, it is difficult to fix up results and continue program execution at the point of an arithmetic trap. Software can force a trap to be continued more easily without the need for complicated fixup code. This is accomplished by specifying any valid

qualifier combination that includes the /S qualifier with each such instruction and following a set of code-generation restrictions in the code that could cause arithmetic traps, allowing those traps to be completed by an OS completion handler.

The AND of all the exception completion qualifiers for trapping instructions is provided to the OS completion handler in the exception summary SWC bit. If SWC is set, a completion handler may find the trigger instruction by scanning backward from the trap PC until each register in the register write mask has been an instruction destination. The trigger instruction is the last instruction in I-stream order to get a trap before the trap shadow. If the SWC bit is clear, no fixup is possible.

- A synchronous trap occurs at the completion of the operation that caused the exception. No instructions can be issued between the completion of the operation that caused the exception and the trap.

24.1.2 Interrupts

The processor arbitrates interrupt requests. When the interrupt priority level (IPL) of an outstanding interrupt is greater than the current IPL, the processor raises IPL to the level of the interrupt and dispatches to entInt, the interrupt entry to the OS. Interrupts are serviced in kernel mode on the kernel stack. Interrupts can come from one of five sources: interprocessor interrupts, I/O devices, the clock, performance counters, or machine checks.

24.2 Processor Status

The processor status (PS) is a four-bit register that contains the current mode (PS<mode>) in bit <3> and a three-bit interrupt priority level (PS<IPL>) in bits <2...0>. The PS<mode> bit is zero for kernel mode and one for user mode. The PS<IPL> bits are always zero if the mode is user and can be zero to 7 if the mode is kernel. The PS is changed when an interrupt or exception is initiated and by the rti, retsys, and swipl instructions.

The uses of the PS values are shown in Table 24–1.

Table 24–1: Processor Status Summary

PS<mode>	PS<IPL>	Mode	Use
1	0	User	User software
0	0	Kernel	System software
0	1	Kernel	System software
0	2	Kernel	System software
0	3	Kernel	Low priority device interrupts
0	4	Kernel	High priority device interrupts
0	5	Kernel	Clock, and interprocessor interrupts
0	6	Kernel	Real-time devices
0	6	Kernel	Correctable error reporting
0	7	Kernel	Machine checks

24.3 Stack Frames

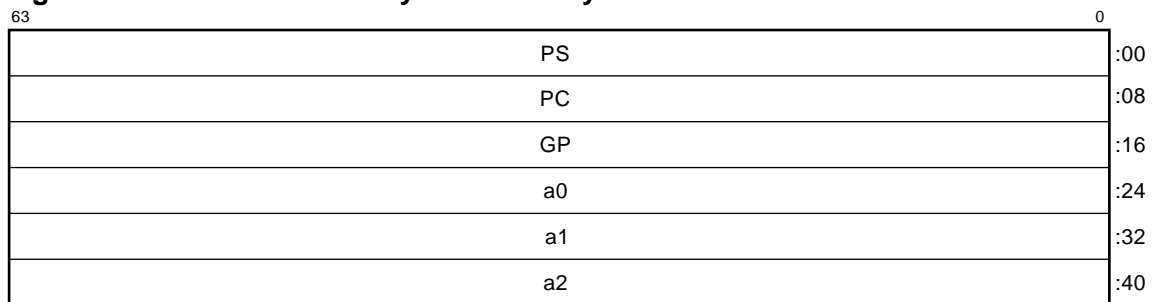
There are three types of system entries: entries for the `callsys` instruction from user mode, entries for exceptions and interrupts from kernel mode, and entries for interrupts from user mode.

Those three types of system entries use one of two stack frame layouts, as follows.

Entries for the `callsys` instruction from user mode, and entries for exceptions and interrupts from kernel mode use the same stack frame layout, as shown in Figure 24–1. The stack frame contains space for the PC, the PS, the saved GP, and the saved registers `a0`, `a1`, `a2`. On entry, the SP points to the saved PS.

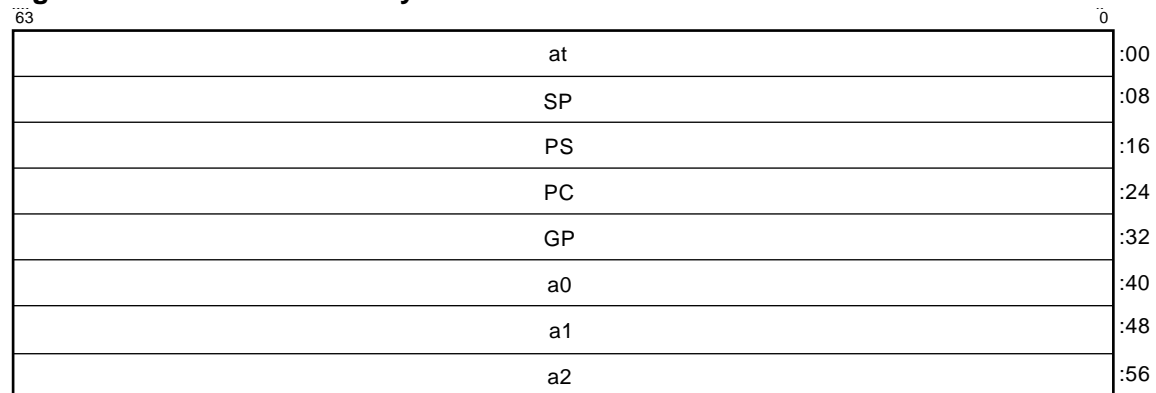
The `callsys` entry saves the PC, the PS, and the GP. The exception and interrupt entries save the PC, the PS, the GP, and also save the registers `a0...a2`.

Figure 24–1 Stack Frame Layout for `callsys` and `rti`



Entries for interrupts from user mode use the stack frame layout as shown in Figure 24–2. The stack frame must be aligned on a 64-byte boundary and contains the registers, `at`, `SP`, `PS`, `PC`, `GP`, and saved registers `a0`, `a1`, and `a2`.

Figure 24–2 Stack Frame Layout for `urti`



24.4 System Entry Addresses

All system entries are in kernel mode. The interrupt priority PS bits (PS<IPL>) are set as shown in the following table. The system entry point address is set by the wrent instruction, as described in Section 21.2.15.

Table 24–2 Entry Point Address Registers

Entry Point	Value in a0	Value in a1	Value in a2	PS<IPL>
entArith	Exception summary	Register mask	UNPREDICTABLE	Unchanged
entIF	Fault or trap type code	UNPREDICTABLE	UNPREDICTABLE	Unchanged
entInt	Interrupt type	Vector	Interrupt parameter	Priority of interrupt
entMM	VA	MMCSR	Cause	Unchanged
entSys	p0	p1	p2	Unchanged
entUna	VA	Opcode	Src/Dst	Unchanged

24.4.1 System Entry Arithmetic Trap (entArith)

The arithmetic trap entry, entArith, is called when an arithmetic trap occurs. On entry, a0 contains the exception summary register and a1 contains the exception register write mask. Section 24.4.1.1 describes the exception summary register and Section 24.4.1.2 describes the register write mask.

24.4.1.1 Exception Summary Register

The exception summary register, shown in Figure 24–3 and described in Table 24–3, records the various types of arithmetic exceptions that can occur together.

Figure 24–3 Exception Summary Register

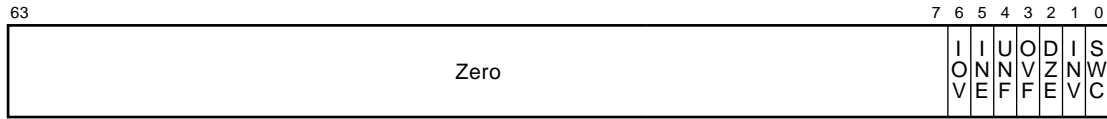


Table 24–3 Exception Summary Register Bit Definitions

Bit	Description
63–7	Zero.
6	<p>Integer overflow (IOV)</p> <p>An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision.</p> <p>An IOV trap is reported for any integer operation whose true result exceeds the destination register size. Integer overflow trap enable can be specified in each arithmetic integer operate instruction and each floating-point convert-to-integer instruction. If integer overflow occurs, the result register is written with the truncated true result.</p>
5	<p>Inexact result (INE)</p> <p>A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.</p> <p>An INE trap is reported if the rounded result of an IEEE operation is not exact. Inexact result trap enable can be specified in each IEEE floating-point operate instruction. The rounded result value is stored in all cases.</p>
4	<p>Underflow (UNF)</p> <p>A floating arithmetic or conversion operation underflowed the destination exponent.</p> <p>An UNF trap is reported when the destination’s smallest finite number exceeds in magnitude the non-zero rounded true result. Floating underflow trap enable can be specified in each floating-point operate instruction. If underflow occurs, the result register is written with a true zero.</p>
3	<p>Overflow (OVF)</p> <p>A floating arithmetic or conversion operation overflowed the destination exponent.</p> <p>An OVF trap is reported when the destination’s largest finite number is exceeded in magnitude by the rounded true result. Floating overflow traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.</p>

Table 24–3 Exception Summary Register Bit Definitions (Continued)

Bit	Description
2	<p>Division by zero (DZE)</p> <p>An attempt was made to perform a floating divide operation with a divisor of zero.</p> <p>A DZE trap is reported when a finite number is divided by zero. Floating divide by zero traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.</p>
1	<p>Invalid operation (INV)</p> <p>An attempt was made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values were illegal.</p> <p>An INV trap is reported for most floating-point operate instructions with an input operand that is an IEEE NaN, IEEE infinity, or IEEE denormal.</p> <p>Floating invalid operation traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.</p>
0	<p>Software completion (SWC)</p> <p>Is set when all of the other arithmetic exception bits were set by floating-operate instructions with the /S qualifier set. See Section 4.7.7.3 for rules about setting the /S qualifier in code that may cause an arithmetic trap, and Section 24.1.1 for rules about using the SWC bit in a trap handler.</p>

24.4.1.2 Exception Register Write Mask

The exception register write mask parameter records all registers that were targets of instructions that set the bits in the exception summary register. There is a one-to-one correspondence between bits in the register write mask quadword and the register numbers. The quadword, starting at bit 0 and proceeding right to left, records which of the registers r0 through r31, then f0 through f31, received an exceptional result.

Note:

For a sequence such as:

```

ADDF    F1, F2, F3
MULF    F4, F5, F3

```

if the add overflows and the multiply does not, the OVF bit is set in the exception summary, and the F3 bit is set in the register mask, even though the overflowed sum in F3 can be overwritten with an in-range product by the time the trap is taken. (This code violates the destination reuse rule for exception completion. See Section 4.7.7.3 for the destination reuse rules.)

The PC value saved in the exception stack frame is the virtual address of the next instruction. This is defined as the virtual address of the first instruction not executed after the trap condition was recognized.

24.4.2 System Entry Instruction Fault (entIF)

The instruction fault or synchronous trap entry is called for bpt, bugchk, gentrap, and opDec synchronous traps, and for a FEN fault (floating-point instruction when the floating-point unit is disabled, FEN EQ 0). On entry, a0 contains a 0 for a bpt, a 1 for bugchk, a 2 for gentrap, a 3 for FEN fault, and a 4 for opDec. No additional data is passed in a1...a2. The saved PC at (SP+00) is the address of the instruction that caused the fault for FEN faults. The saved PC at (SP+00) is the address of the instruction after the instruction that caused the bpt, bugchk, gentrap, and opDec synchronous traps.

24.4.3 System Entry Hardware Interrupts (entInt)

The interrupt entry is called to service a hardware interrupt or a machine check. Table 24–4 shows what is passed in a0...a2 and the PS<IPL> setting for various interrupts.

Table 24–4 System Entry Hardware Interrupts

Entry Type	Value in a0	Value in a1	Value in a2	PS<IPL>
Interprocessor interrupt	0	UNPREDICTABLE	UNPREDICTABLE	5
Clock	1	UNPREDICTABLE	UNPREDICTABLE	5
Correctable error	2	Interrupt vector	Pointer to Logout Area	7
Machine check	2	Interrupt vector	Pointer to Logout Area	7
I/O device interrupt	3	Interrupt vector	UNPREDICTABLE	Level of device
Performance counter	4	Interrupt vector	UNPREDICTABLE	6

On entry to the hardware interrupt routine, the IPL has been set to the level of the interrupt. For hardware interrupts, register a1 contains a platform-specific interrupt vector. That platform-specific interrupt vector is typically the same value as the SCB offset value that would be returned if the platform was running OpenVMS PALcode.

For a correctable error or machine check interrupt, a1 contains a platform-specific interrupt vector and a2 contains the kseg address of the platform-specific logout area. The interrupt vector value and logout area format are typically the same as those used by the platform when running OpenVMS PALcode.

The machine check error summary (MCES) register, shown in Figure 24–4 and described in Table 24–5, records the correctable error and machine check interrupts in progress.

Figure 24–4 Machine Check Error Status (MCES) Register

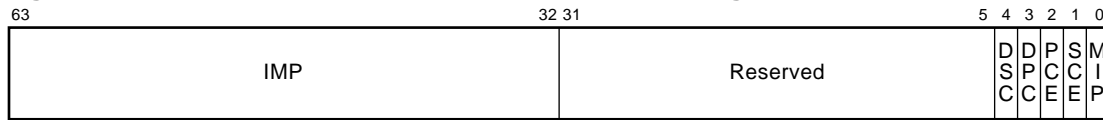


Table 24–5 Machine Check Error Status (MCES) Register Bit Definitions

Bit	Symbol	Description
63–32		IMP.
31–5		Reserved.
4	DSC	Disable system correctable error in progress. Set to disable system correctable error reporting.
3	DPC	Disable processor correctable error in progress. Set to disable processor correctable error reporting.
2	PCE	Processor correctable error in progress. Set when a processor correctable error is detected. Should be cleared by the processor correctable error handler when the logout frame may be reused.
1	SCE	System correctable error in progress. Set when a system correctable error is detected. Should be cleared by the system correctable error handler when the logout frame may be reused.
0	MIP	Machine check in progress. Set when a machine check occurs. Must be cleared by the machine check handler when a subsequent machine check can be handled. Used to detect double machine checks.

The MIP flag in the MCES register is set prior to invoking the machine check handler. If the MIP flag is set when a machine check is being initiated, a double machine check halt is initiated instead. The machine check handler needs to clear the MIP flag when it can handle a new machine check.

Similarly, the SCE or PCE flag in the MCES register is set prior to invoking the appropriate correctable error handler. That error handler should clear the appropriate correctable error in progress when the logout area can be reused by hardware or PALcode. PALcode does not overwrite the logout area.

Correctable processor or system error reporting may be suppressed by setting the respective DPC or DSC flag in the MCES register. When the DPC or DSC flag is set, the corresponding error is corrected, but no correctable error interrupt is generated.

24.4.4 System Entry MM Fault (entMM)

The memory-management fault entry is called when a memory management exception occurs. On entry, a0 contains the faulting virtual address and a1 contains the MMCSR (see Section 22.9). On entry, a2 is set to a minus one (–1) for an instruction fetch fault, to a plus one (+1) for a fault caused by a store instruction, or to a 0 for a fault caused by a load instruction.

24.4.5 System Entry Call System (entSys)

The system call entry is called when a callsys instruction is executed in user mode. On entry, only registers (t8...t11) have been modified. The PC+4 of the callsys instruction, the user global pointer, and the current PS are saved on the kernel stack. Additional space for a0...a2 is allocated. After completion of the system service routine, the kernel code executes a CALL_PAL retsys instruction.

24.4.6 System Entry Unaligned Access (entUna)

The unaligned access entry is called when a load or store access is not aligned. On entry, a0 contains the faulting virtual address, a1 contains the zero extended six-bit opcode (bits <31:26>) of the faulting instruction, and a2 contains the zero extended data source or destination register number (bits <25:21>) of the faulting instruction.

24.5 PALcode Support

24.5.1 Stack Writeability and Alignment

PALcode only accesses the kernel stack. Any PALcode accesses to the kernel stack that would produce a memory-management fault will result in a kernel-stack-not-valid halt. The stack pointer must always point to a quadword-aligned address. If the kernel stack is not quadword aligned on a PALcode access, a kernel-stack-not-valid halt is initiated.

Console Interface Architecture (III)

This part describes an architected console interface and contains the following chapters:

- Chapter 25, Console Subsystem Overview (III)
- Chapter 26, Console Interface to Operating System Software (III)
- Chapter 27, System Bootstrapping (III)

Chapter 25

Console Subsystem Overview (III)

On an Alpha system, underlying control of the system platform hardware is provided by a console. The console:

- Initializes, tests, and prepares the system platform hardware for Alpha system software.
- Bootstraps (loads into memory and starts the execution of) system software.
- Controls and monitors the state and state transitions of each processor in a multiprocessor system in the absence of operating system control.
- Provides services to system software that simplify system software control of and access to platform hardware.
- Provides a means for a "console operator" to monitor and control the system.

The console interacts with system platform hardware to accomplish the first three tasks. The mechanisms of these interactions are specific to the platform hardware; however, the net effects are common to all systems. Chapter 27 describes these functions.

The console interacts with system software once control of the system platform hardware has been transferred to that software. Chapter 26 discusses the basic functions of a console and its interaction with Alpha system software.

The console interacts with the console operator through a virtual display device or console terminal. The console operator may be a person or a management application. The console terminal forms the interface between the console and a console presentation layer. The functions of that presentation layer and the display formats are described in Section 25.3.

An Alpha multiprocessor system has one primary processor and one or more secondary processors. The primary processor:

- Can legally refer to the console I/O devices
- Can legally send characters to the console terminal
- Can legally receive characters from the console terminal
- Has direct access to a BB_WATCH on the system
- Is named in response to an inquiry as to which processor is primary

All other processors in the system are secondary processors.

25.1 Console Implementations

The implementation of an Alpha console varies from system to system. Regardless of implementation, the console on each system provides the functionality described in this chapter and in Chapters 2 and 3. The console may be implemented as:

- "Embedded," or co-resident in the hardware platform complex that contains the processors
- "Detached," or resident on a separate hardware platform
- Any hybrid of the above

The distinction is somewhat arbitrary. A detached console may have cooperating special code that executes on one of the processors; an embedded console may have a cooperating management application that executes on a remote machine.

Regardless of the actual implementation, each console must provide:

- A virtual display device, the default "console terminal."
This device allows the console operator to issue commands and receive displays. With no hardware errors and with the proper console-lock setting, the default console terminal device provides reliable communication with the rest of the console.
- Reliable access to console functionality by system software and the console operator.
All console functionality must appear to reside within the console at all times. All console functions must be accessible in a timely manner, without prior notification, and reliably.
- Secure communications with system software and the console operator.
All console communication paths must be able to be made secure by either physical measures or encryption methods.
- A mechanism by which the console can gain control of a processor that is executing system software.
This mechanism must preserve the execution state of system software; it must be possible for the console to gain control of the processor and subsequently continue system software execution successfully.

Note:

Continuation of system software by the CONTINUE command may be restricted to the early stages of booting for hardware configurations where the console keyboard is connected by way of USB.

- A mechanism that locks the console.
A console lock prohibits the user from accessing a selected subset (or all) of console functions. The console lock may be a console password, a key switch, jumper, or any other implementation-specific mechanism. The lock is either "locked" or "unlocked."

25.2 Console Implementation Registry

This chapter, and Chapters 26 and 27, specify required console functions. Some of these functions have attributes that may vary with console implementation; consoles may also provide more than the required functions. Console functions or attributes that may vary with implementation include:

- Supported console terminal blocks (CTBs)
- Supported environment variables
- Environment variable value formats, such as `BOOT_DEV` or `BOOT_OSFLAGS`
- Configuration data block format
- Supported callback routines
- Supported bootstrap media
- Implementation-specific HALT codes or messages

The goal of the Alpha console architecture is to promote a consistent interface across all Alpha systems. Some console functionality is inherently implementation specific and cannot be required of all Alpha systems; some may be applicable to more than one Alpha system. To prevent the proliferation of interfaces and achieve commonality of function whenever possible, the Alpha console architecture requires that:

- Any console function that is visible to system software and is not specified by these chapters must be registered with the Alpha architecture group.
- Any console function that is visible to an on-site or remote console operator (including Field Service engineers) and is not specified by these chapters must be registered with the Alpha architecture group.
- Whenever possible, implementations must use previously registered functions rather than inventing new variations.

Console functions intended for use solely by development engineering or expert-level repair and diagnosis are excluded from these requirements.

25.3 Console Presentation Layer

The following functions are assumed to be provided in the console presentation layer:

- `BOOT` (bootstrap the system)
- `CONTINUE` (continue execution)
- `STARTCPU` (start a given secondary)
- `INITIALIZE` (initialize system)
- `INITIALIZECPU` (initialize a given processor)
- `HALTCPU` (force a given processor into console I/O mode)
- `HALTCRASH` (cause a given processor to initiate a crash)

25.4 Messages

The console generates a binary message code to the console presentation layer to signal messages, such as audit trail or error messages. The console presentation layer interprets the binary code into something that is meaningful to the console operator.

25.5 Security

The means by which the console achieves a secure communications path with system software and with the console operator is implementation specific. Embedded consoles have the built-in capability of secure communications with system software. Detached consoles can achieve this security by residing in the same room as the Alpha system and communicating with it over a private connection. Detached consoles can also achieve security by using an encrypted protocol over a shared connection. This latter method allows a workstation over a network to function as the console.

25.6 Internationalization

Wherever possible, console implementations should support the goals of internationalization:

- Each message has a binary message code. The console presentation layer interprets the code into a meaningful message display of the appropriate language and characters.
- Consoles should avoid explicitly interpreting character set encoding (such as ISO Latin-1). Character strings are to be viewed as simple byte strings. Thus, the GETC console callback routine supports from one-to-four-byte character encodings, depending on the currently selected language and character set; the PUTS routine outputs only a byte stream.
- ASCII strings are used in certain fields of the HWRPB and certain interprocessor communications due to COMPAQ STD 12 and to present a common interface to system software.
- The currently selected character set encoding and language to be used for the console terminal are defined by the CHAR_SET and LANGUAGE environment variables.
- The end of a character string passed between the console and the operating system as an argument to a console callback routine is determined by passing its length.
- Console callback routines should be written to be independent from character set encoding and language. At a minimum, every implementation must support ISO Latin-1 character set encodings, which requires the following properties:
 1. The GETC console callback routine returns a one byte character (see Section 26.3.4).
 2. The PROCESS_KEYCODE console callback routine returns a one-byte character (see Section 26.3.4).
 3. English console presentation layers are strongly encouraged to use the actual values as defined in Table 26-6, rather than creating aliases.

Other supported character set encodings are determined by platform product requirements.

- The console presentation layer is independent of the required console functionality interface.

25.7 Documentation Note

Chapters 25 through 27 apply to the OpenVMS, Tru64 UNIX, and Alpha Linux operating systems. The few functional descriptions that are unique to one operating system are described as such. However, because of contextual equivalence in this section and in the interests of brevity, any text concerning the OpenVMS hardware privileged context block (HWPCB) applies equally to the Tru64 UNIX and Alpha Linux privileged context block (PCB).

Console Interface to Operating System Software (III)

This chapter describes the interactions between the console subsystem and system software. These services depend on state that is shared between the console and system software. Shared state is contained in the Hardware Restart Parameter Block (HWRPB) and a number of environment variables. The HWRPB is a data structure that is directly accessed by both the console and system software; the environment variables are indirectly accessed by system software. Specifically:

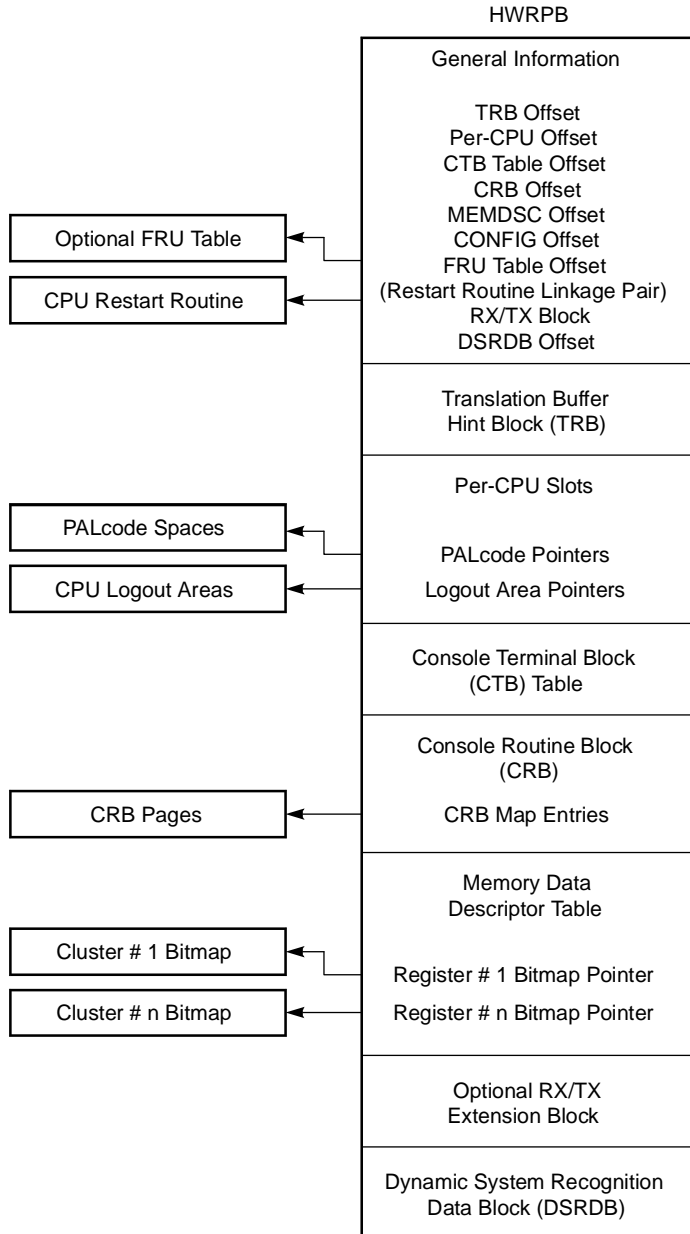
- Section 26.1 describes the HWRPB.
- Section 26.2 describes the environment variables.
- Section 26.3 describes the service, or callback, routines provided by the console to system software.
- Section 26.4 describes the communication between the console and system software.

26.1 Hardware Restart Parameter Block (HWRPB)

The Hardware Restart Parameter Block (HWRPB) is a page-aligned data structure that is shared between the console and system software. The HWRPB is a critical resource during bootstraps, powerfail recoveries, and other restart situations. An overview of the HWRPB is shown in Figure 26–1. The individual HWRPB fields are shown in Figure 26–2 and described in Table 26–1.

The console creates the HWRPB and the required per-CPU, CTB, CRB, MEMDSC, and DSRDB offset blocks as a physically contiguous structure during console initialization. Fields within the HWRPB and the required offset blocks are updated by the console and system software during and after system bootstrapping. The console must be able to locate the HWRPB and the required offset blocks at all times. Neither the console nor system software may move the HWRPB or the required offset blocks to different physical memory locations; subsequent operation of the system is UNDEFINED if such an attempt is made.

Figure 26–1 HWRPB Overview



The HWRPB and the required offset blocks must comprise a virtually contiguous structure at all times. Before transferring control to system software, the console maps the HWRPB and the required offset blocks into contiguous addresses beginning at virtual address 0000 0000 1000 0000₁₆ in the initial bootstrap address space. If system software subsequently changes this virtual mapping, any new mapping must preserve the relative offsets of all fields and blocks; all physically contiguous pages must remain virtually contiguous. Some of the data structures located by HWRPB fields need not be contiguous with the HWRPB. The structures that may be discontinuous are the PALcode spaces, the logout areas, the CRB pages, the FRU table, and the tested memory bitmaps located by the MEMDSC table.

All offset blocks must be at least quadword aligned. The starting address of an offset block is determined by adding the contents of the HWRPB offset field to the starting address of the HWRPB. For example, the starting address of the MEMDSC block is given by:

$$\begin{aligned}\text{MEMDSC Address} &= \text{HWRPB address} + \text{MEMDSC OFFSET} \\ &= \text{HWRPB address} + (\text{HWRPB}[200])\end{aligned}$$

The total size of the HWRPB and the required offset blocks is on the order of 8K bytes to 16K bytes. The size is contained in the HWRPB_SIZE field at HWRPB[24]. The required offset blocks may be offset from the HWRPB in any order; the HWRPB offset fields must not be used to infer the size of the HWRPB or any offset block.

Figure 26–2: Hardware Restart Parameter Block Structure

63	32 31	0
Physical Address of the HWRPB		:HWRPB
"HWRPB"		:+08
HWRPB Revision		:+16
HWRPB Size		:+24
Primary CPU ID		:+32
Page Size (Bytes)		:+40
Number of Extension VA Bits	Number of PA Bits	:+48
Maximum Valid ASN		:+56
System Serial Number (SSN)		:+64
System Type		:+80
System Variation		:+88
System Revision		:+96
Interval Clock Interrupt Frequency		:+104
Cycle Counter Frequency		:+112
Virtual Page Table Base		:+120
Reserved for Architecture Use		:+128
Offset to Translation Buffer Hint Block		:+136
Number of Processor Slots		:+144
Per-CPU Slot Size		:+152
Offset to Per-CPU Slots		:+160
Number of CTBs		:+168
CTB Size		:+176
Offset to Console Terminal Block Table		:+184
Offset to Console Callback Routine Block		:+192
Offset to Memory Data Descriptor Table		:+200
Offset to Configuration Data Block (If Present)		:+208
Offset to FRU Table (If Present)		:+216
Virtual Address of Terminal Save State Routine		:+224
Procedure Value of Terminal Save State Routine		:+232
Virtual Address of Terminal Restore State Routine		:+240
Procedure Value of Terminal Restore State Routine		:+248

Figure 26-2 : Hardware Restart Parameter Block Structure (Continued)

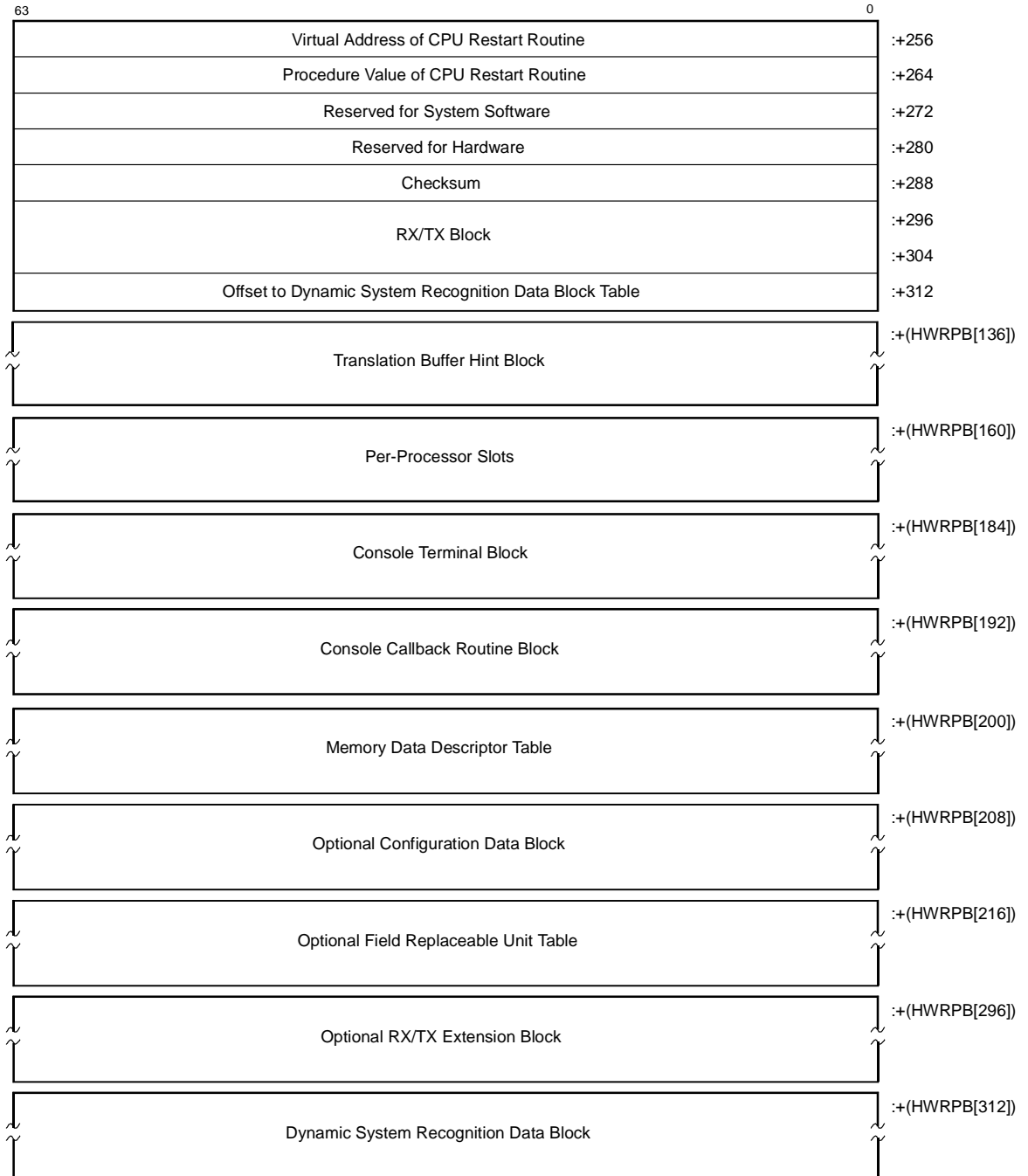


Table 26–1 HWRPB Fields

Offset	Description
HWRPB	HWRPB PA ¹ Starting physical address of the HWRPB field. This field is used by the console to validate the HWRPB.
+08	HWRPB VALIDATION ¹ Quadword containing "HWRPB<0><0><0>" (0000 0042 5052 5748 ₁₆). This field is used by the console to validate the HWRPB.
+16	HWRPB REVISION ¹ Format of the HWRPB. See Section 26.1.1. The HWRPB revision level for this version of the architecture specification is 13.
+24	HWRPB SIZE ¹ Size in bytes of the HWRPB and required physically contiguous TBB, per-CPU, CTB, CRB, MEMDSC, RX/TX Extension, and DSRDB offset blocks. The size of the FRU table is included if the table is physically contiguous with the HWRPB and the required offset blocks. Unsigned field.
+32	PRIMARY CPU ID ^{1,3} WHAMI of the primary processor. System software modifies this field only at primary switch; see Section 27.5.6. Unsigned field.
+40	PAGE SIZE ¹ Number of bytes within a page for this Alpha processor implementation. Unsigned field.
+48	PA SIZE ¹ Size of the physical address space in bits for this Alpha processor implementation. PA SIZE must be 48 bits or less. Unsigned 32-bit field.
+52	EXTENDED VA SIZE ² If this processor implementation supports mixed 48-bit/43-bit VA mode and the processor is running in mixed mode, field is set to 5; otherwise, field is set to zero. Unsigned 32-bit field.
+56	MAX VALID ASN ¹ Maximum ASN value allowed by this Alpha processor implementation. Unsigned field.
+64	SYSTEM SERIAL NUMBER ¹ Full COMPAQ STD 12 serial number for this Alpha system. This octaword field contains a 10-character ASCII serial number determined at the time of manufacture; see COMPAQ STD 12 for format information. See Section 26.1.1.1.
+80	SYSTEM TYPE ¹ Family or system hardware platform. See Section 26.1.1. Unsigned field.

Table 26–1 HWRPB Fields (Continued)

Offset	Description
+88	SYSTEM VARIATION ^{1,3} Subtype variation of the system. This may include the member of the system family and whether the system has optional features such as multiprocessor support or special power supply conditioning. See Sections 26.1.1 and 26.1.1.2 for optional features.
+96	SYSTEM REVISION CODE ¹ COMPAQ STD 12 revision field for this Alpha system. Four ASCII characters. See Section 26.1.1.1.
+104	INTERVAL CLOCK INTERRUPT FREQUENCY ¹ Number of interval clock interrupts per second (scaled by 4096) in this Alpha system. Interrupts occur only if enabled. Unsigned field.
+112	CYCLE COUNTER FREQUENCY ¹ Number of SCC and PCC updates per second for the primary CPU in this Alpha system. See the RPCC instruction (Section 4.11.9) and, for OpenVMS, the CALL_PAL RSCC instruction (Section 10.1.12). Unsigned field.
+120	VIRTUAL PAGE TABLE BASE ^{2,3} Virtual address of the base of the entire page table structure. The console sets this field at system bootstraps and restores the virtual page table base register (pointer) with this value at all processor restarts. System software is responsible for updating this field whenever the virtual page table base register (pointer) is modified. See Sections 27.4.1.3, 27.4.3.5, and 27.5.1.
+128	Reserved for architecture use; SBZ.
+136	TB HINT OFFSET ¹ Unsigned offset to the starting address of the Translation Buffer Hint Block (TBB). See Section 26.1.2.
+144	NUMBER OF PER-CPU SLOTS ¹ Number of per-CPU slots present. See Section 26.4 for constraints on the maximum value that may be stored here. See Section 26.1.3 for the per-CPU slot format. Unsigned field.
+152	PER-CPU SLOT SIZE ¹ Size in bytes of each per-CPU slot rounded up to the next integer multiple of 128. See Section 26.1.3. Unsigned field.
+160	CPU SLOT OFFSET ¹ Unsigned offset to the first per-CPU slot in the HWRPB. See Section 26.1.3.

Table 26–1 HWRPB Fields (Continued)

Offset	Description
+168	NUMBER OF CTB ¹ Number of Console Terminal Blocks (CTBs) contained in the CTB table. See Section 26.3.8.2. Unsigned field.
+176	CTB SIZE ¹ Size in bytes of the largest Console Terminal Block (CTB) contained in the CTB table. See Section 26.3.8.2. Unsigned field.
+184	CTB OFFSET ¹ Unsigned offset to the starting address of the Console Terminal Block (CTB) table. See Section 26.3.8.2.
+192	CRB OFFSET ¹ Unsigned offset to the starting address of the Console Callback Routine Block (CRB). See Section 26.3.8.1.
+200	MEMDSC OFFSET ¹ Unsigned offset to the starting address of the Memory Data Descriptor Table (MEMDSC). See Sections 26.1.5 and 27.4.1.1.
+208	CONFIG OFFSET ¹ Unsigned offset to the starting address of the Configuration Data Table (CONFIG). If zero, no CONFIG table exists. See Section 26.1.4.
+216	FRU TABLE OFFSET ¹ Unsigned offset to the starting address of the Field Replaceable Unit Table (FRU). If zero, no FRU table exists. See Sections 26.1.5 and 27.4.1.1.
+224	SAVE_TERM RTN VA ^{2,3} Starting virtual address of a routine that saves console terminal state. This routine is optionally provided by system software. See Section 27.5.7. Set to zero by the console at system bootstraps.
+232	SAVE_TERM VALUE ^{2,3} Procedure value of the SAVE_TERM routine optionally provided by system software. The console copies this value into R27 before invoking the routine. See Section 27.5.7. Set to zero by the console at system bootstraps.
+240	RESTORE_TERM RTN VA ^{2,3} Starting virtual address of a routine that restores console terminal state. This routine is optionally provided by system software. See Section 27.5.7. Set to zero by the console at system bootstraps.

Table 26–1 HWRPB Fields (Continued)

Offset	Description
+248	RESTORE_TERM VALUE ^{2,3} Procedure value of the RESTORE_TERM routine optionally provided by system software. The console copies this value into R27 before invoking the routine. See Section 27.5.7. Set to zero by the console at system bootstraps.
+256	RESTART RTN VA ^{2,3} Starting virtual address of a CPU restart routine provided by system software. The console restarts system software by transferring control to this routine. See Section 27.5. Set to zero by the console at system bootstraps.
+264	RESTART VALUE ^{2,3} Procedure value of the CPU restart routine provided by system software. During the restart process, the console copies this value into R27 before transferring control to the CPU restart routine. See Section 27.5. Set to zero by the console at system bootstraps.
+272	RESERVED FOR SYSTEM SOFTWARE ^{2,3} Reserved for use by system software. Set to zero by the console at system bootstraps.
+280	RESERVED FOR HARDWARE ¹ Reserved for use by hardware.
+288	HWRPB CHECKSUM ^{2,3} Checksum of all the quadwords of the HWRPB from offset [00] to [280], inclusive. Computed as a 64-bit sum, ignoring overflows. Used to validate the HWRPB during warm bootstraps, restarts, and secondary starts. Set by console initialization; recomputed and updated whenever a HWRPB field with offset [00] to [280], inclusive, is modified by the console or system software.
+296	RX/TX BLOCK Receive/transmit control block. Interpreted as shown in Table 26–2 and described in Section 26.4. Two unsigned quadwords.
+312	DSRDB OFFSET ¹ Unsigned offset to the starting address of the Dynamic System Recognition Data Block.
+(HWRPB[136])	TB HINT BLOCK ^{2,3} Quadword-aligned block that describes the characteristics of the translation buffer (TB) granularity hints. See Section 26.1.2.
+(HWRPB[160])	Per-CPU SLOTS ^{2,3} 128 byte-aligned slots that describe each processor in the system. See Section 26.1.3.

Table 26–1 HWRPB Fields (Continued)

Offset	Description
+(HWRPB[184])	CTB TABLE ¹ Quadword-aligned Console Terminal Block Table. Set at console initialization; modified by console terminal callbacks. See Section 26.3.8.2.
+(HWRPB[192])	CONSOLE CALLBACK ROUTINE BLOCK ^{2,3} Quadword-aligned block that describes the location and mapping of the console callback routines. Set at system bootstrap; modified by console FIXUP callback. See Section 26.3.8.1.
+(HWRPB[200])	MEMDSC ^{1,3} Quadword-aligned Memory Data Descriptor Table. Set at console initialization; preserved across warm bootstraps. See Sections 26.1.5 and 27.4.1.1.
+(HWRPB[208])	CONFIG BLOCK ¹ Optional implementation-dependent configuration block. See Section 26.1.4.
+(HWRPB[216])	FRU TABLE ^{1, 3} Optional implementation-dependent field replaceable unit table. This table may contain distributed memory descriptors. See Sections 26.1.5 and 27.4.1.1.
+(HWRPB[296])	RX/TX EXTENSION BLOCK Optional receive/transmit extension block. See Table 26–2 and Section 26.4.
+(HWRPB[312])	DSRDB ¹ Quadword-aligned Dynamic System Recognition Data Block (DSRDB).

¹ Initialized by the console at cold system bootstrap only. Preserved unchanged by the console at all warm system bootstraps.

² Initialized by the console at all system bootstraps (cold or warm).

³ May be modified by system software.

26.1.1 Serial Number, Revision, Type, and Variation Fields

The HWRPB contains several serial number, revision, type, and variation fields that describe the Alpha system platform hardware and PALcode. System software uses these fields to identify hardware-dependent support code that must be loaded or enabled. These fields are examined early in operating system bootstrap; if one of the fields contains a value that is unrecognized or incompatible with the operating system, the bootstrap attempt fails. Diagnostic software uses these fields to guide field installation and upgrade procedures and for material and parts control.

In multiprocessor systems, the processor type and PALcode revisions need not be identical for all processors. Console and system software can use these fields to determine if multiprocessor operation is viable. This evaluation may be performed by the running primary, the starting secondary, or a combination of both. For an example, see Section 27.4.3.3.

26.1.1.1 Serial Number and Revision Fields

The revision fields include:

- HWRPB revision — HWRPB[16]

This field identifies the format of the HWRPB. Since the HWRPB is shared between the console and system software, both must agree on the field offsets, formats, and interpretations.

- System serial number and revision — HWRPB[64] and HWRPB[96]

These fields identify the system platform hardware serial number and revision according to COMPAQ STD 12.

The system serial number and revision fields must be distinct from the processor serial number and revision fields in the per-CPU table, pointed to by HWRPB[152]. In particular, on multiprocessing systems, the system fields must not be simply replicated from the fields of the primary processor. The system fields must be constant regardless of which processor serves as primary and must have persistence across processor failures and/or replacement.

- Processor type and processor variation (capabilities) — SLOT[176] and SLOT[184]

These per-CPU slot fields identify each Alpha processor and its capabilities. The type field (SLOT[176]) contains a major and minor subfield. The major subfield identifies the processor family and the minor subfield identifies the particular membership in that family.

The variation (capabilities) field (SLOT[184]) identifies any system-specific attributes (such as local memory or cache size)

Processor type and variation field assignments are listed in Appendix D.

- Processor Revision — SLOT[192]

This per-CPU slot field identifies the processor hardware revision according to COMPAQ STD 12.

- PALcode Revision — SLOT[168]

This field identifies the PALcode revision required and/or in use by the processor. System software uses the PALcode variation and PALcode compatibility subfields. The variation subfield indicates whether the PALcode image includes extensions or functional variations necessary to a given operating system or application.

Programming Note:

For example, a PALcode variation may contain a different TB fill routine. System software (and optionally the console) uses the compatibility subfield to ensure that all processors in a multiprocessor system are using compatible PALcode images.

PALcode revisions are specific to the system platform and processor major type. The file name of distributed PALcode images must contain sufficient information to distinguish the intended system platform and processor.

- PALcode Revisions Available — SLOT[464]

This field identifies the PALcode variant revisions that have been previously loaded on this processor. System software uses these fields to determine if a given PALcode

variant and revision are present before PALcode switching. The format follows the PALcode revision field in SLOT[168].

PALcode variation assignments are listed in Appendix D.

26.1.1.2 System Type and Variation Fields

The system type and system variation fields are HWRPB[80] and HWRPB[88].

These fields identify the Alpha system platform. System software infers attributes such as physical address offsets and I/O device locations from the system type. The system type field contains the family and member identification numbers, along with the major and minor sub-field identifiers. It is described in Appendix D. The system variation field is described in Table 26–2.

Hardware platforms that belong to the same family must use the same major and minor SRM console revision values.

The system variations are defined in Table 26–2.

Table 26–2 System Variation Field (HWRPB[88])

Bits	Description
63 – 34	Additional feature specifications — These bits are defined within the context of ECOs.
33	RX/TX EXTENT — Indicates how the RXRDY/TXRDY bitmasks are implemented. If clear, the RX/TX Block at HWRPB+296 contains a 64-bit RXRDY bitmask and a 64-bit TXRDY bitmask, and no RX/TX Extension Block exists. If set, the RX/TX Block at HWRPB+296 contains an offset from the beginning of the HWRPB to the RX/TX Extension Block. See Section 26.4.
32	Separate Page Table Structures. If set, support for the Virtual Address Boundary (VIRBND) register exists.
31 – 24	Reserved — MBZ
23 – 16	Platform-specific variations. Registered values to be provided by system and platform representatives.
15 – 10	System Type Specific (STS). Registered system identifiers for system member identification.
9	GRAPHICS — If set, indicates that the platform contains an embedded graphics processor. Initialized by the console at all cold bootstraps.
8	POWERFAIL RESTART — If set, indicates that the console should restart all available processors on a powerfail recovery. If clear, only the primary processor will be restarted. Cleared by the console at system bootstraps; may be set by system software.

Table 26–2 System Variation Field (HWRPB[88]) (Continued)

Bits	Description										
7 – 5	POWERFAIL — Indicates the type of powerfail (if any) implemented by this platform. See Section 27.5.3 for more information. Defined values include:										
	<table border="1"> <thead> <tr> <th><7:5></th> <th>Interpretation</th> </tr> </thead> <tbody> <tr> <td>000</td> <td>Reserved</td> </tr> <tr> <td>001</td> <td>United</td> </tr> <tr> <td>010</td> <td>Separate</td> </tr> <tr> <td>011</td> <td>Full battery backup of system platform hardware</td> </tr> </tbody> </table>	<7:5>	Interpretation	000	Reserved	001	United	010	Separate	011	Full battery backup of system platform hardware
<7:5>	Interpretation										
000	Reserved										
001	United										
010	Separate										
011	Full battery backup of system platform hardware										
	Initialized by the console at all cold bootstraps.										
4 – 1	CONSOLE — Indicates the type of console. Defined values include:										
	<table border="1"> <thead> <tr> <th><4:1></th> <th>Interpretation</th> </tr> </thead> <tbody> <tr> <td>0000</td> <td>Reserved</td> </tr> <tr> <td>0001</td> <td>Detached service processor</td> </tr> <tr> <td>0010</td> <td>Embedded console</td> </tr> <tr> <td>Other</td> <td>Reserved for future use</td> </tr> </tbody> </table>	<4:1>	Interpretation	0000	Reserved	0001	Detached service processor	0010	Embedded console	Other	Reserved for future use
<4:1>	Interpretation										
0000	Reserved										
0001	Detached service processor										
0010	Embedded console										
Other	Reserved for future use										
	Initialized by the console at all cold bootstraps.										
0	MPCAP — If set, indicates this system platform is capable of being configured as a multiprocessor; all support for multiprocessing is present, even if only one processor is present. If clear, this system supports a uniprocessor only. Initialized by the console at all cold bootstraps.										

26.1.2 Translation Buffer Hint Block

The Translation Buffer Hint Block (TBB) contains information on the characteristics of the instruction stream translation buffer (ITB) and data stream translation buffer (DTB) granularity hints (GH). All processors in a multiprocessor Alpha system must implement the same granularity hints. The granularity hint fields are listed in Table 26–3.

Implementation Note:

The granularity hint fields described in Table 26–3 have not been implemented in any Alpha console through the 21364/EV7x.

The TBB consists of 8 quadwords, 4 for each of the translation buffers (ITB and DTB). The 4 quadwords contain 16 word fields; each word contains the number of entries in the translation buffer that implement a combination of granularity hints (including none).

Table 26–3: Granularity Hint Fields

Offset ₁₆	Granularity Hint
0	None
2	1 page
4	8 pages
6	1 and 8 pages
8	64 pages
A	1 and 64 pages
C	8 and 64 pages
E	1, 8, and 64 pages
10	512 pages
12	1 and 512 pages
14	8, and 512 pages
16	1, 8, and 512 pages
18	64 and 512 pages
1A	1, 64, and 512 pages
1C	8, 64, and 512 pages
1E	1, 8, 64, and 512 pages

26.1.3 Per-CPU Slots in the HWRPB

Information on the state of a processor is contained in a "per-CPU slot" data structure for that processor. The per-CPU slots form a contiguous array indexed by CPU ID. The starting address of the first per-CPU slot is given by the offset HWRPB[160] relative to the starting address of the HWRPB. The number of per-CPU slots is given in HWRPB[144]. Each per-CPU slot must be 128-byte-aligned to ensure natural alignment of the hardware privileged context block (HWPCB) at SLOT[0]. The slot size, rounded up to the nearest multiple of 128 bytes, is given in HWRPB[152].

CPU IDs are determined by the implementation. The only requirement is that they be in the range of zero to the maximum number of processors the particular platform supports minus one.

Software Note:

OpenVMS supports CPU IDs in the range 0–31 only.

Each per-CPU slot contains information necessary to bootstrap, start, restart or halt the processor. The format is shown in Figure 26–3 and described in Table 26–4. The hardware privileged context block (HWPCB) specifies the context in which the loaded system software will execute.

The console must initialize the per-CPU slot for the primary processor before system bootstrap. The per-CPU slot fields for secondary processors are set by a combination of the console and system software. The console updates the halt information at error halts and before processor restarts.

Slots corresponding to nonexistent processors are zeroed. There may be more per-CPU slots than are necessary in any given Alpha system. A system implementation may reserve HWRPB space for processors that are not present at system bootstrap.

An Alpha system may support internally different, yet software compatible, PALcode for different processors in a multiprocessor implementation. Each per-CPU slot contains a PALcode memory descriptor that locates the PALcode used by that processor. See Section 27.3.1 for information on PALcode loading and initialization on the primary processor and Section 27.4.3.3 for information on PALcode loading and initialization on secondary processors.

The starting address of a per-CPU slot is calculated by:

$$\begin{aligned} \text{Slot Address} &= \{\text{CPU ID} * \text{slot size}\} + \text{offset} + \text{HWRPB base} \\ &= \{\text{CPU ID} * \text{HWRPB}[152]\} + \text{HWRPB}[160] + \#\text{HWRPB} \end{aligned}$$

The address may be physical or virtual.

Figure 26–3 Per-CPU Slot in HWRPB

~	Bootstrap/Restart HWPCB	~:SLOT
	Per-CPU State Flag Bits	:+128
	PALcode Memory Length	:+136
	PALcode Scratch Length	:+144
	Physical Address of PALcode Memory Space	:+152
	Physical Address of PALcode Scratch Space	:+160
	PALcode Revision Required by Processor	:+168
	Processor Type	:+176
	Processor Variation	:+184
	Processor Revision	:+192
	Processor Serial Number	:+200
	Physical Address of Logout Area	:+216
	Logout Area Length	:+224
	Halt PCBB	:+232
	Halt PC	:+240
	Halt PS	:+248
	Halt Argument List (R25)	:+256
	Halt Return Address (R26)	:+264
	Halt Procedure Value (R27)	:+272
	Reason for Halt	:+280
	Reserved for Software	:+288
~	Interprocessor Console Buffer Area	~:+296
~	PALcode Revisions Available Block	~:+464
	Processor Software Compatibility Field	:+592
	Console Data Log Physical Address	:+600
	Console Data Log Length	:+608
	Cache Information	:+616
	Cycle Counter Frequency	:+624
~	Reserved for Architecture Use	~:+632

Table 26–4 Per-CPU Slot Fields

Offset	Description
SLOT	HWPCB ^{1,2} Hardware privileged context block (HWPCB) for this processor. See Table 27–10 for the contents as set by the console.
+128	STATE FLAGS ^{1,2} Current state of this processor. See Table 26–5 for the interpretation of each bit.
+136	PALCODE MEMORY SPACE LENGTH ^{3,4,5} Number of bytes required by this processor for PALcode memory. Unsigned field.

Table 26–4 Per-CPU Slot Fields (Continued)

Offset	Description														
+144	PALCODE SCRATCH SPACE LENGTH ^{3,4,5} Number of bytes required by this processor for PALcode scratch space. Unsigned field.														
+152	PA OF PALCODE MEMORY SPACE ^{2,3,5} Starting physical address of PALcode memory space for this processor. PALcode memory space must be page aligned. See Section 27.3.1 or Section 27.4.3.3.														
+160	PA OF PALCODE SCRATCH SPACE ^{2,3,5} Starting physical address of PALcode scratch space for this processor. PALcode scratch space must be page aligned. See Section 27.3.1 or Section 27.4.3.3.														
+168	PALCODE REVISION ^{2,3,4,6} PALcode revision level for this processor:														
	<table border="1"> <thead> <tr> <th>Bits</th> <th>Interpretation</th> </tr> </thead> <tbody> <tr> <td>63 – 48</td> <td>Maximum number of processors that can share this PALcode image</td> </tr> <tr> <td>47 – 32</td> <td>PALcode compatibility (0–65535): 0 Unknown 1–65535 Compatibility revision</td> </tr> <tr> <td>31 – 24</td> <td>SBZ</td> </tr> <tr> <td>23 – 16</td> <td>PALcode variation (0–255)</td> </tr> <tr> <td>15 – 8</td> <td>PALcode major revision (0–255)</td> </tr> <tr> <td>7 – 0</td> <td>PALcode minor revision (0–255)</td> </tr> </tbody> </table>	Bits	Interpretation	63 – 48	Maximum number of processors that can share this PALcode image	47 – 32	PALcode compatibility (0–65535): 0 Unknown 1–65535 Compatibility revision	31 – 24	SBZ	23 – 16	PALcode variation (0–255)	15 – 8	PALcode major revision (0–255)	7 – 0	PALcode minor revision (0–255)
Bits	Interpretation														
63 – 48	Maximum number of processors that can share this PALcode image														
47 – 32	PALcode compatibility (0–65535): 0 Unknown 1–65535 Compatibility revision														
31 – 24	SBZ														
23 – 16	PALcode variation (0–255)														
15 – 8	PALcode major revision (0–255)														
7 – 0	PALcode minor revision (0–255)														
	This field identifies the PALcode revision required by the console and/or processor initialization. The major and minor PALcode revisions are set at console initialization; the remaining fields are set during PALcode loading and initialization. This field must be updated after PALcode switching to reflect the new PALcode environment. See Sections 26.1.1 and Section 27.4.3.3. Also see														
+176	PROCESSOR TYPE ^{3,4} Type of this processor:														
	<table border="1"> <thead> <tr> <th>Bits</th> <th>Interpretation</th> </tr> </thead> <tbody> <tr> <td>63 – 32</td> <td>Minor type</td> </tr> <tr> <td>31 – 0</td> <td>Major type</td> </tr> </tbody> </table>	Bits	Interpretation	63 – 32	Minor type	31 – 0	Major type								
Bits	Interpretation														
63 – 32	Minor type														
31 – 0	Major type														

The processor types are defined in Appendix D.

Table 26–4 Per-CPU Slot Fields (Continued)

Offset	Description										
+184	<p>PROCESSOR VARIATION^{3,4}</p> <p>The following processor variations are defined:</p> <table border="1"> <thead> <tr> <th>Bit</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>63–3</td> <td>RESERVED — MBZ</td> </tr> <tr> <td>2</td> <td>PRIMARY ELIGIBLE (PE) — If set, indicates that this processor is eligible to become a primary processor. The processor has direct access to the console, a BB_WATCH, and all I/O devices. See Chapter 27.</td> </tr> <tr> <td>1</td> <td>IEEE-FP — If set, indicates this processor supports IEEE floating-point operations and data types. If clear, this processor has no such support.</td> </tr> <tr> <td>0</td> <td>VAX-FP — If set, indicates this processor supports VAX floating-point operations and data types. If clear, this processor has no such support.</td> </tr> </tbody> </table>	Bit	Description	63–3	RESERVED — MBZ	2	PRIMARY ELIGIBLE (PE) — If set, indicates that this processor is eligible to become a primary processor. The processor has direct access to the console, a BB_WATCH, and all I/O devices. See Chapter 27.	1	IEEE-FP — If set, indicates this processor supports IEEE floating-point operations and data types. If clear, this processor has no such support.	0	VAX-FP — If set, indicates this processor supports VAX floating-point operations and data types. If clear, this processor has no such support.
Bit	Description										
63–3	RESERVED — MBZ										
2	PRIMARY ELIGIBLE (PE) — If set, indicates that this processor is eligible to become a primary processor. The processor has direct access to the console, a BB_WATCH, and all I/O devices. See Chapter 27.										
1	IEEE-FP — If set, indicates this processor supports IEEE floating-point operations and data types. If clear, this processor has no such support.										
0	VAX-FP — If set, indicates this processor supports VAX floating-point operations and data types. If clear, this processor has no such support.										
+192	<p>PROCESSOR REVISION^{3,4}</p> <p>Full COMPAQ STD 12 revision field for this processor. This quadword field contains four ASCII characters. See Section 26.1.1.</p>										
+200	<p>PROCESSOR SERIAL NUMBER^{3,4}</p> <p>Full DEC STD serial number for this processor module. This octaword field contains a 10-character ASCII serial number determined at the time of manufacture; see COMPAQ STD 12 for format information.</p>										
+216	<p>PA OF LOGOUT AREA^{3,4}</p> <p>Starting physical address of PALcode logout area for this processor. Logout areas must be at least quadword aligned.</p>										
+224	<p>LOGOUT AREA LENGTH^{3,4}</p> <p>Number of bytes in the PALcode logout area for this processor.</p>										
+232	<p>HALT PCBB^{1,7}</p> <p>Value of the PCBB register when a processor halt condition is encountered by this processor. Initialized to the address of the hardware privileged context block (HWPCB) at offset [0] from this per-CPU slot at system bootstraps or secondary processor starts.</p>										
+240	<p>HALT PC^{1,7}</p> <p>Value of the PC when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.</p>										
+248	<p>HALT PS^{1,7}</p> <p>Value of the PS when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.</p>										

Table 26–4 Per-CPU Slot Fields (Continued)

Offset	Description																						
+256	<p>HALT ARGUMENT LIST^{1,7}</p> <p>Value of R25 (argument list) when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.</p>																						
+264	<p>HALT RETURN ADDRESS^{1,7}</p> <p>Value of R26 (return address) when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.</p>																						
+272	<p>HALT PROCEDURE VALUE^{1,7}</p> <p>Value of R27 (procedure value) when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.</p>																						
+280	<p>REASON FOR HALT^{1,7}</p> <p>Indicates why this processor was halted. Values include:</p> <table border="1"> <thead> <tr> <th>Code₁₆</th> <th>Reason</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Bootstrap, processor start, or powerfail restart</td> </tr> <tr> <td>1</td> <td>Console operator requested a system crash</td> </tr> <tr> <td>2</td> <td>Processor halted due to kernel-stack not-valid halt</td> </tr> <tr> <td>3</td> <td>Invalid SCBB</td> </tr> <tr> <td>4</td> <td>Invalid PTBR</td> </tr> <tr> <td>5</td> <td>Processor executed CALL_PAL HALT instruction in kernel mode</td> </tr> <tr> <td>6</td> <td>Double error abort encountered</td> </tr> <tr> <td>7</td> <td>Machine check while in PALcode environment</td> </tr> <tr> <td>8 – FFF</td> <td>Reserved</td> </tr> <tr> <td>Other</td> <td>Implementation-specific</td> </tr> </tbody> </table> <p><i>Code</i> is set to "0" at console initialization.</p>	Code ₁₆	Reason	0	Bootstrap, processor start, or powerfail restart	1	Console operator requested a system crash	2	Processor halted due to kernel-stack not-valid halt	3	Invalid SCBB	4	Invalid PTBR	5	Processor executed CALL_PAL HALT instruction in kernel mode	6	Double error abort encountered	7	Machine check while in PALcode environment	8 – FFF	Reserved	Other	Implementation-specific
Code ₁₆	Reason																						
0	Bootstrap, processor start, or powerfail restart																						
1	Console operator requested a system crash																						
2	Processor halted due to kernel-stack not-valid halt																						
3	Invalid SCBB																						
4	Invalid PTBR																						
5	Processor executed CALL_PAL HALT instruction in kernel mode																						
6	Double error abort encountered																						
7	Machine check while in PALcode environment																						
8 – FFF	Reserved																						
Other	Implementation-specific																						
+288	<p>RESERVED FOR SOFTWARE²</p> <p>Reserved for use by system software. Zeroed at system bootstraps or secondary processor starts.</p>																						
+296	<p>RXTX BUFFER AREA</p> <p>Used for interprocessor console communication. See Section 26.4.</p>																						

Table 26–4 Per-CPU Slot Fields (Continued)

Offset	Description														
+464	<p>PALCODE AVAILABLE^{3,4}</p> <p>Block of 16 quadwords that list previously loaded PALcode variations that are available to the console or operating system for PALcode switching.</p> <p>The first offset (SLOT[464]) is reserved for an overall firmware revision field for this processor, the format of which is determined by the HWRPB revision level found at HWRPB[16]. If HWRPB[16] contains 6 or less, the format for SLOT[464] is platform specific. If HWRPB[16] is greater than 6, the format for SLOT[464] is as follows:</p> <table border="1"> <thead> <tr> <th>Bits</th> <th>Interpretation</th> </tr> </thead> <tbody> <tr> <td>63–48</td> <td>Maximum number of processors that can share this console</td> </tr> <tr> <td>47–32</td> <td>Console build sequence number (0–16383)</td> </tr> <tr> <td>31–24</td> <td>SBZ</td> </tr> <tr> <td>23–16</td> <td>Variant (0 for console version)</td> </tr> <tr> <td>15–8</td> <td>Console major revision (0–255)</td> </tr> <tr> <td>7–0</td> <td>Console minor revision (0–255)</td> </tr> </tbody> </table> <p>The format of each subsequent quadword follows the PALcode revision field (SLOT[168]). Each quadword is indexed by PALcode variant. If the quadword is non-zero, the PALcode variant has been loaded and the operating system may switch to that PALcode variant by passing the variant number to CALL_PAL SWPPAL.</p>	Bits	Interpretation	63–48	Maximum number of processors that can share this console	47–32	Console build sequence number (0–16383)	31–24	SBZ	23–16	Variant (0 for console version)	15–8	Console major revision (0–255)	7–0	Console minor revision (0–255)
Bits	Interpretation														
63–48	Maximum number of processors that can share this console														
47–32	Console build sequence number (0–16383)														
31–24	SBZ														
23–16	Variant (0 for console version)														
15–8	Console major revision (0–255)														
7–0	Console minor revision (0–255)														
+592	<p>PROCESSOR SOFTWARE COMPATIBILITY FIELD⁸</p> <p>Type of pre-existing processor that is software compatible with existing processor. Format follows SLOT[176].</p> <table border="1"> <thead> <tr> <th>Bits</th> <th>Interpretation</th> </tr> </thead> <tbody> <tr> <td>63–32</td> <td>Minor type</td> </tr> <tr> <td>31–0</td> <td>Major type</td> </tr> </tbody> </table>	Bits	Interpretation	63–32	Minor type	31–0	Major type								
Bits	Interpretation														
63–32	Minor type														
31–0	Major type														

Table 26–4 Per-CPU Slot Fields (Continued)

Offset	Description
+600	Console Data Log Physical Address Physical address of in-memory buffer of console data to be passed to the operating system (if any, otherwise zero).
+608	Console Data Log Length Length in bytes of console data (if any, otherwise zero).
+616	Description of the cache that implements that level in the memory hierarchy of most significance to software page coloring techniques, and management by buffering of large data sets. Bits Interpretation 63:56 Degree of set associativity, expressed in the following values: Value Meaning 0 Fully associative cache 1 Direct-mapped cache <i>n</i> Number <i>n</i> of sets in the cache 55:48 Cache characteristics mask, as follows: Bits Interpretation 55:49 Reserved for future use 48 Set for write-back; clear for write-through 47:32 Size of an individual cache block, expressed as the log base–2 value of the block’s size in bytes. 31:0 Total size of the cache in units of KBytes.
+624	Cycle Counter Frequency Number of SCC and PCC updates per second for this CPU. If this field is zero, the cycle counter frequency is found at HWRPB[112] and that value should be used.

- ¹ Initialized by the console for the primary at all system bootstraps (cold or warm) and for a secondary before processor start.
- ² May be modified by system software for a secondary before processor start.
- ³ Initialized by the console for a secondary at cold system bootstrap only. Preserved unchanged by the console at all other times.
- ⁴ Initialized by the console for the primary at cold system bootstrap only. Preserved unchanged by the console at all other times.
- ⁵ Support PALcode loading as described in Section 27.3.
- ⁶ May be modified by system software for the primary.
- ⁷ Set by the console at all processor halts.
- ⁸ Initialized by the console at cold bootstrap and never written by system software or console.

Table 26–5 Per-CPU State Flags

Bit	Description																
63:24	RESERVED; MBZ.																
23:16	<p>HALT REQUESTED^{1,2,3}</p> <p>Indicates the console action requested by system software executing on this processor. Values include:</p> <table border="1"> <thead> <tr> <th>Code₁₆</th> <th>Reason</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Default (no specific action)</td> </tr> <tr> <td>1</td> <td>SAVE_TERM/RESTORE_TERM exit</td> </tr> <tr> <td>2</td> <td>Cold bootstrap requested</td> </tr> <tr> <td>3</td> <td>Warm bootstrap requested</td> </tr> <tr> <td>4</td> <td>Remain halted (no restart)</td> </tr> <tr> <td>5</td> <td>System power-off requested; requires at least HWRPB revision 8.</td> </tr> <tr> <td>Other</td> <td>Reserved</td> </tr> </tbody> </table>	Code ₁₆	Reason	0	Default (no specific action)	1	SAVE_TERM/RESTORE_TERM exit	2	Cold bootstrap requested	3	Warm bootstrap requested	4	Remain halted (no restart)	5	System power-off requested; requires at least HWRPB revision 8.	Other	Reserved
Code ₁₆	Reason																
0	Default (no specific action)																
1	SAVE_TERM/RESTORE_TERM exit																
2	Cold bootstrap requested																
3	Warm bootstrap requested																
4	Remain halted (no restart)																
5	System power-off requested; requires at least HWRPB revision 8.																
Other	Reserved																
	<p>Set to "0" at system bootstraps and secondary processor starts. May be set to non-zero by system software before processor halt and subsequent processor entry into console I/O mode. See Sections 27.5.7 and 27.4.5.</p>																
15:9	RESERVED; MBZ.																
8	<p>PALCODE LOADED (PL)^{3,4,5}</p> <p>Indicates that this processor's PALcode image has been loaded into the address given in the processor's slot PALcode memory space address field. See Sections 27.3.1 and 27.4.3.3.</p>																
7	<p>PALCODE MEMORY VALID (PMV)^{3,4,5}</p> <p>Indicates that this processor's PALcode memory and scratch space addresses are valid. Set after the necessary memory is allocated and the addresses are written into the processor's slot. See Sections 27.3.1 and 27.4.3.3.</p>																
6	<p>PALCODE VALID (PV)^{4,5}</p> <p>Indicates that this processor's PALcode is valid. Set after PALcode has been successfully loaded and initialized. See Sections 27.3.1 and 27.4.3.3.</p>																
5	<p>CONTEXT VALID (CV)^{1,3}</p> <p>Indicates that the HWPCB in this slot is valid. Set after the console or system software initializes the HWPCB in this slot. See Sections 27.3.1 and 27.4.3.</p>																
4	<p>OPERATOR HALTED (OH)^{1,6}</p> <p>Indicates that this processor is in console I/O mode as the result of explicit operator action. See Section 27.5.8.</p>																
3	<p>PROCESSOR PRESENT (PP)^{4,5}</p> <p>Indicates that this processor is physically present in the configuration.</p>																

Table 26–5 Per-CPU State Flags (Continued)

Bit	Description
2	PROCESSOR AVAILABLE (PA) ^{4,5} Indicates that this processor is available for use by system software. The PA bit may differ from the PP bit based on self-test or other diagnostics, or as the result of a console command that explicitly sets this processor unavailable.
1	RESTART CAPABLE (RC) ^{1,2,3,6} Indicates that system software executing on this processor is capable of being restarted if a detected error halt, powerfail recovery, or other error condition occurs. Cleared by the console and set by system software. See Sections 27.4.1.3, 27.4.3.6, and 27.5.1.
0	BOOTSTRAP IN PROGRESS (BIP) ^{1,2,3} For the primary, this bit indicates that this processor is undergoing a system bootstrap. For a secondary, this bit indicates that a CPU start operation is in progress. Set by the console and cleared by system software. See Sections 27.4.1.3, 27.4.3.6, and 27.5.1.

¹ Initialized by the console for the primary at all system bootstraps (cold or warm) and for a secondary before processor start.

² May be modified by system software for the primary.

³ May be modified by system software for a secondary before processor start.

⁴ Initialized by the console for primary at cold system bootstrap only. Preserved unchanged by the console at all other times.

⁵ Initialized by the console for a secondary at cold system bootstrap only. Preserved unchanged by the console at all other times.

⁶ Set by the console at all processor halts.

26.1.4 Configuration Data Block

Systems may have a Configuration Data Block (CONFIG). The format of the block and whether it exists in a system is implementation specific. If present, the block must be mapped in the bootstrap address space. The CONFIG offset at HWRPB[208] contains the block offset address; if no CONFIG block exists, the offset is zero. The first quadword of a CONFIG block must contain the size in bytes of the block. The second quadword must contain a checksum for the block. The checksum is computed as a 64-bit sum, ignoring overflows, of all quadwords in the configuration data block except the checksum quadword.

26.1.5 Field Replaceable Unit Table

Systems may have a field replaceable unit (FRU) table. The format of the table and whether it exists in a system is implementation specific. If present and physically contiguous to the HWRPB and the required offset blocks, the table must be mapped in the bootstrap address space and its size included in the HWRPB SIZE field.

The FRU table offset at HWRPB[216] contains the physical offset from the base of the HWRPB; if no FRU table exists, the offset is zero. Comparing the offset value to the HWRPB SIZE indicates whether or not the FRU table is contiguous to the HWRPB and the required offset blocks.

As of HWRPB Revision 12, a system may choose to distribute memory descriptors such that they are contained within the structure located by the FRU table offset at HWRPB[216]. See Section 27.4.1.1.

26.2 Environment Variables

The environment variables provide an easily extensible mechanism for managing complex console state. Such state may be variable length, may change with system software, may change as a result of console state changes, and may be established by the console presentation layer. Environment variables may be read, written, or saved.

An environment variable consists of an identifier (ID) and a byte stream value maintained by the console. There are three classes of environment variables:

1. Common to all implementations: ID = 0 to $3F_{16}$.

These have meaning to both the console and system software. All consoles must implement all of these environment variables.

2. Specific to a given console implementation: ID = 40 to $7F_{16}$.

These have meaning to a given console implementation and system software implementation. Support for these environment variables is optional.

3. Specific to system software: ID = 80 to FF_{16} .

These have meaning to a given system software application or implementation; the console passes these environment variables between the console presentation layer and the target application without interpretation. Support for these environment variables is optional.

If a console supports optional environment variables, they must be described in the relevant console implementation specification and registered with the Alpha architecture group.

The value, format, and size of each environment variable depends on the environment variable and the console implementation. The size of an environment variable value is specified in bytes. The byte stream value of most environment variables consists of an ASCII string.

The booting environment variables, `BOOT_DEV`, `BOOTDEF_DEV`, and `BOOTED_DEV`, contain values that can consist of multiple fields and lists. For those variables, the values are parsed as follows:

- Each field is delimited by one and only one space " " (20_{16}).
- Each list element is delimited by one and only one comma ", " ($2C_{16}$).
- Any numeric quantities are expressed in decimal.
- All characters are case-blind and may be expressed in uppercase or lowercase.

Other examples of environment variables that have list values are `BOOTED_OSFLAGS` and `DUMP_DEV`.

Programming Note:

For example, `BOOT_DEV` might consist of "0 4 MSCP,0 1 MOP" and `BOOT_OSFLAGS` might consist of "7,2,28".

System software uses the console environment variable routines to access the environment variables. Each environment variable is identified by an identification number (ID). If the console resolves the ID, the associated byte stream value is returned. The console environment variable routines present system software with a consistent interface to environment variables regardless of the presentation layer and internal console representation. The console operator interacts with the console presentation layer to access environment variables. See Section 25.3 for details.

In a multiprocessor system, the console must ensure that the dynamic state created by the environment variables is common to all processors. It must not be possible for a value observed on a secondary to differ from that observed on the primary or another secondary. This is necessary to support bootstrapping, restarting a processor, and switching the primary.

Some environment variables contain critical state that must be maintained across console initializations and system power transitions. Other environment variables contain dynamic state that must be initialized at console initialization and retained across warm bootstraps. Still others contain dynamic state that is initialized at each system bootstrap.

Environment variable values that must be maintained across console initializations must be retained in some sort of nonvolatile storage. Default values for these environment variables must be set before system shipment. Thus, there are three possible values: the dynamic value, the default value retained in nonvolatile storage, and the initial default value set in nonvolatile storage before system shipment. The console need not preserve the initial default value. If console implementation preserves the initial default value, that value is accessible only to the console presentation layer; system software accesses only the dynamic and default (last written) values. The dynamic and default values may differ at any time after console initialization as the result of changes by system software or the console operator.

The internal mechanisms for representing and implementing environment variables are determined by the console and are unknown to both system software and the console presentation layer. The method of handling the required nonvolatile storage also depends on the implementation.

Table 26–6 lists the environment variables maintained by the console. Each environment ID is also assigned a symbolic name that is used to reference the environment variable elsewhere in this specification. Tables 26–7 and 26–8, respectively, list supported languages and character sets.

Table 26–6 Required Environment Variables

Environment Variable		Description
ID ₁₆	Symbol	
00		Reserved
01	AUTO_ACTION ^{1,2}	<p>Console action following an error halt or powerup. Defined values and the action invoked are:</p> <ul style="list-style-type: none"> • "BOOT" (544F 4F42₁₆) bootstrap • "HALT" (544C 4148₁₆) halt • "RESTART" (54 5241 5453 4552₁₆) restart <p>Any other value causes a halt; The default value when the system is shipped is "HALT". See Section 27.1.1.</p>
02	BOOT_DEV ²	<p>Device list used by the last (or currently in progress) bootstrap attempt. The console modifies BOOT_DEV at console initialization and when a bootstrap attempt is initiated by a BOOT command. The value of BOOT_DEV is set from the device list specified with the BOOT command or, if no device list is specified, BOOTDEF_DEV. The console uses BOOT_DEV without change on all bootstrap attempts that are not initiated by a BOOT command. See Section 27.4.1.5. The format is independent of the console presentation layer.</p>
03	BOOTDEF_DEV ^{1,2}	<p>Device list from which bootstrapping is to be attempted when no path is specified by a BOOT command. See Section 27.4.1.5. The format follows BOOT_DEV. The default value when the system is shipped indicates a valid implementation-specific device or NULL (00₁₆).</p>
04	BOOTED_DEV ³	<p>Device used by the last (or currently in progress) bootstrap attempt. Value is one of the devices in the BOOT_DEV list. See Section 27.4.1.5. The format is independent of the console presentation layer.</p>
05	BOOT_FILE ^{1,2}	<p>File name to be used when a bootstrap requires a file name and when the bootstrap is not the result of a BOOT command or when no file name is specified on a BOOT command. The console passes the value between the console presentation layer and system software without interpretation; the value is preserved across warm bootstraps. The default value when the system is shipped is NULL (00₁₆).</p>

Table 26–6 Required Environment Variables (Continued)

Environment Variable ID ₁₆	Symbol	Description
06	BOOTED_FILE ³	File name used by the last (or currently in progress) bootstrap attempt. The value is derived from BOOT_FILE or the current BOOT command. The console passes the value between the console presentation layer and system software without interpretation.
07	BOOT_OSFLAGS ^{1,2}	Additional parameters to be passed to system software when the bootstrap is not the result of a BOOT command or when no parameters are specified on a BOOT command. The console preserves the value across warm bootstraps and passes the value between the console presentation layer and system software without interpretation. The default value when the system is shipped is NULL (00 ₁₆).
08	BOOTED_OSFLAGS ³	Additional parameters passed to system software during the last (or currently in progress) bootstrap attempt. The value is derived from BOOT_OSFLAGS or the current BOOT command. The console passes the value between the console presentation layer and system software without interpretation.
09	BOOT_RESET ^{1,2}	Indicates whether a full system reset is performed in response to an error halt or BOOT command. Defined values and the action invoked are: <ul style="list-style-type: none">• "OFF" (46 464F₁₆) warm bootstrap, no full system reset is performed.• "ON" (4E4F₁₆) cold bootstrap, a full system reset is performed. See Sections 27.4.1 and 27.4.2. The default value when the system is shipped is implementation specific.
0A	DUMP_DEV ^{1,2}	Device used to write operating system crash dumps. The format follows BOOTED_DEV and is independent of the console presentation layer. The value is preserved across warm bootstraps. The default value when the system is shipped indicates an implementation-specific device or NULL (00 ₁₆).
0B	ENABLE_AUDIT ^{1,2}	Indicates whether audit trail messages are to be generated during bootstrap. Defined values and the action invoked are: <ul style="list-style-type: none">• "OFF" (46 464F₁₆). Audit trail messages suppressed.• "ON" (4E4F₁₆). Audit trail messages generated. The default value when the system is shipped is "ON."
0C	LICENSE ^{1,3}	Software license in effect. The value is derived in an implementation-specific manner during console initialization.

Table 26–6 Required Environment Variables (Continued)

Environment Variable ID₁₆	Symbol	Description
0D	CHAR_SET ^{1,2}	Current console terminal character set encoding. Defined values are given in Table 26–8. The default value when the system is shipped is determined by the manufacturing site.
0E	LANGUAGE ^{1,2}	Current console terminal language. Defined values are given in Table 26–7. The default value when the system is shipped is determined by the manufacturing site.
0F	TTY_DEV ^{1,2,3}	Current console terminal unit. Indicates which entry of the CTB table corresponds to the actual console terminal. The value is preserved across warm bootstraps. The default value is "0" (30 ₁₆).
10– 3F		Reserved for Compaq.
40 – 7F		Reserved for console implementation use.
80 – FF		Reserved for system software use.

¹ Nonvolatile. The last value saved by system software or set by console commands is preserved across system initializations, cold bootstraps, and long power outages.

² Warm nonvolatile. The last value set by system software is preserved across warm bootstraps and restarts.

³ Read-only. The variable cannot be modified by system software or console commands.

Table 26–7 Supported Languages

LANGUAGE₁₆	Language	Character Set	GETC Bytes
0	None (cryptic)	ISO Latin–1	1
30	Dansk	ISO Latin–1	1
32	Deutsch	ISO Latin–1	1
34	Deutsch (Schweiz)	ISO Latin–1	1
36	English (American)	ISO Latin–1	1
38	English (British/Irish)	ISO Latin–1	1
3A	Espanol	ISO Latin–1	1
3C	Francais	ISO Latin–1	1
3E	Francais (Canadian)	ISO Latin–1	1
40	Francais (Suisse Romande)	ISO Latin–1	1
42	Italiano	ISO Latin–1	1
44	Nederlands	ISO Latin–1	1
46	Norsk	ISO Latin–1	1
48	Portugues	ISO Latin–1	1

Table 26–7 Supported Languages (Continued)

LANGUAGE ₁₆	Language	Character Set	GETC Bytes
4A	Suomi	ISO Latin–1	1
4C	Svenska	ISO Latin–1	1
4E	Vlaams	ISO Latin–1	1
Other	Reserved		

Table 26–8 Supported Character Sets

CHAR_SET ₁₆	Character Set
0	ISO Latin–1
Other	Reserved

26.3 Console Callback Routines

System software can access certain system hardware components through a set of callback routines provided by the Alpha console. These routines give system software an architecturally consistent and relatively simple interface to those components.

All of the console callback routines may be used by system software when the operating system has only restricted functionality, such as during bootstrap or crash. When invoked in this context, the console may assume full control of system platform hardware. Some of the console callback routines may be used by system software when the operating system is fully functional. Such usage imposes constraints on the console implementation.

All routines must be called by system software executing in kernel mode. All routines require that the HWRPB and the per-CPU, CTB, and CRB offset blocks are virtually mapped and kernel read/write accessible. If these conditions are not met, the results are UNDEFINED. If conditions from within user mode are not met, the results are UNPREDICTABLE. Some of the routines execute correctly only at or above certain IPLs.

The routines must never modify any processor registers except those explicitly indicated by the routine descriptions.

26.3.1 System Software Use of Console Callback Routines

The console callback routines present an environment to the operating system in which the following behavior must be implemented. These routines must:

- Not alter the current IPL
- Not alter the current execution mode
- Not disable or mask interrupts
- Not alter any registers except as explicitly defined by the routine interface
- Not alter the existing memory management policy
- Not usurp any existing interrupt mechanisms
- Be interruptible

- Ensure timely completion

Once the operating system is bootstrapped, the console must not reclaim resources transferred to that operating system. This includes both the issuing and servicing of I/O device interrupts, interprocessor interrupts, and exceptions.

It is the responsibility of the console implementation to ensure that these console callback routines may be invoked at multiple IPLs, may be interrupted, and may be invoked by multiple system software threads. The operation of these routines must appear to be atomic to the calling system software even if that software thread is interrupted.

In a multiprocessor system, some console routines may be invoked only on the primary processor. A secondary processor may invoke only a subset of these routines and then only under a limited set of conditions. These conditions are explicitly stated in the routine descriptions; if violated, the results are UNDEFINED.

26.3.2 System Software Invocation of Console Callback Routines

With the exception of the FIXUP routine, all of the routines are accessed uniformly through a common DISPATCH procedure. The target routine is identified by a function code. All console callback routines are invoked using the Alpha standard calling conventions.

Any memory management exceptions generated by incorrect mapping or inaccessibility of console callback routine parameters produce UNDEFINED results. This occurs naturally for those console callback routines that are intended for use while the operating system is fully functional; these routines execute in the unmodified context of the operating system.

For those routines intended for use only while the operating system has restricted functionality, the DISPATCH routine must ensure that any conflicts in mapping or accessibility are resolved before permitting the console to gain control of the system platform hardware.

26.3.3 Console Callback Routine Summary

The console callback routines fall into four functional groups:

1. Console terminal interaction
2. Generic I/O device access
3. Environment variable manipulation
4. Miscellaneous

The hexadecimal function code, name, and function for each routine are summarized in Table 26–9.

Table 26–9 Console Callback Routines

Code ₁₆	Name	Function Invoked
Console Terminal Routines		
01	GETC	Get character from console terminal
02	PUTS	Put byte stream to console terminal
03	RESET_TERM	Reset console terminal to default
04	SET_TERM_INT	Set console terminal interrupts

Table 26–9 Console Callback Routines (Continued)

Code₁₆	Name	Function Invoked
05	SET_TERM_CTL	Set console terminal controls
06	PROCESS_KEYCODE	Process and translate keycode
07	CONSOLE_OPEN	Opens the console terminal I/O device for use
08	CONSOLE_CLOSED	Terminates use of the console terminal I/O device
09 – F		Reserved
Console Generic I/O Device Routines		
10	OPEN	Open I/O device for access
11	CLOSE	Close I/O device for access
12	IOCTL	Perform I/O device-specific operations
13	READ	Read I/O device
14	WRITE	Write I/O device
15 – 1F		Reserved
Console Environment Variable Routines		
20	SET_ENV	Set (write) an environment variable
21	RESET_ENV	Reset (default) an environment variable
22	GET_ENV	Get (read) an environment variable
23	SAVE_ENV	Save current environment variables
Console Miscellaneous Routines		
30	PSWITCH	Switch primary processor
(None)	FIXUP	Remap console callback routines
(None)	DISPATCH ¹	Access console callback routine
32	BIOS_EMUL	Run BIOS emulation callback routine
Other		Reserved

¹ DISPATCH is not a callback routine. It is a routine that transfers control to callback routines. DISPATCH accepts the function code of any callback routine as an input parameter and transfers control to the selected callback routine. Therefore, there is no specific section to describe DISPATCH.

All Alpha consoles must implement:

- All console terminal routines except PROCESS_KEYCODE
- All console generic I/O device routines
- All environment variable routines except SAVE_ENV
- The FIXUP and DISPATCH miscellaneous routines

The PSWITCH routine is required for all Alpha multiprocessor systems that support dynamic primary switching. See Section 27.5.6.

26.3.4 Console Terminal Routines

Alpha consoles provide system software with a consistent interface to the console terminal, regardless of the physical realization of that terminal. This interface consists of the console terminal block (CTB) table and a number of console terminal routines. Each CTB contains the characteristics of a terminal device that can be accessed through the console terminal routines; see Section 26.3.8.2.

There is *only one* console terminal. The CTB table may contain multiple CTBs and the console terminal routines may be used to access multiple terminal devices. Each terminal device is identified by a "unit number" that is the index of its CTB within the CTB table. The TTY_DEV environment variable indicates the unit, hence the CTB, of the console terminal. The console terminal unit is determined at system bootstrap and cannot be altered by system software. Console terminal device interrupts are delivered at the console terminal device IPL to the primary processor; interrupts can be redirected to a secondary only when switching the primary processor.

The console terminal routines permit system software to access the console terminal in a device-independent way. These routines may be invoked while the operating system is fully functional as well as during operating system bootstrap or crash. All console terminal routines are subject to the constraints given in Section 26.3.1. These routines must:

- Not alter the current IPL or current mode.

These routines must be invoked in kernel mode at or above the console terminal device IPL.

- Not alter the existing memory management policy.

All internal pointers must have been remapped by FIXUP.

- Not block interrupts.

The operating system must be capable of continuing to receive hardware interrupts at higher IPLs.

- Be interruptible and re-entrant.

These routines may be invoked at multiple IPLs and their execution may be interrupted. However, console terminal callback operations are not necessarily atomic. In the event of re-entrant invocations, it is UNPREDICTABLE whether or not the interrupted operation will fail and characters may be transmitted or received out of order.

The time required for console terminal routines to complete is UNPREDICTABLE; however, a console implementation will attempt to minimize the time whenever possible.

Software Note:

Implementations must limit the execution time to significantly less than the interval clock interrupt period. A return after partial operation completion is preferable to long latency.

When invoking these routines, system software must:

- Be executing in kernel mode at or above the console terminal device IPL.
If these routines are invoked in other modes, their execution causes UNPREDICTABLE operation. If invoked at lower IPLs, their execution causes UNDEFINED operation.
- Be executing on the primary processor in a multiprocessor configuration.
If these routines are invoked on secondary processors in kernel mode, their execution causes UNDEFINED operation.
- Be prepared to service any resulting console terminal interrupts, if enabled.
System software must provide valid interrupt service routines for the console terminal transmit and receive interrupts. The operating system interrupt service routines must be established before enabling interrupts; otherwise the operation of the system is UNDEFINED.

Programming Note:

Any console terminal interrupt service routines established by the console before transferring control to operating system software are not transferred to the operating system nor are they remapped by FIXUP. Any console terminal interrupts will be delivered only after the operating system lowers IPL from the console terminal device IPL.

Implementation Note:

The implementation of console terminal I/O interrupts is specific to the system hardware platform. An example of implementation-specific characteristics is console terminal SCB vectors.

26.3.4.1 GETC — Get Character from Console Terminal

Format:

char = DISPATCH (GETC,unit)

Inputs:

GETC = R16; GETC function code – 01₁₆
unit = R17; Terminal device unit number
retadr = R26; Return address

Outputs:

char = R0; Returned character and status:

R0<63:61>	'000'	Success, character received
	'001'	Success, character received, more to be read
	'100'	Failure, character not yet ready for reception
	'110'	Failure, character received with error
	'111'	Failure, character received with error, more to be read
R0<60:48>		Device-specific error status
R0<47:40>		SBZ
R0<39:32>		Terminal device unit number returning character
R0<31:0>		Character read from console terminal

GETC attempts to read one character from a console terminal device and, if successful, returns that character in R0<31:0>. The character is not echoed on the terminal device. The size of the returned character is from one to four bytes and is a function of the current character set encoding and language (see Table 26–7). The routine performs any necessary keycode mapping.

For implementations that support multiple directly addressable terminal devices, R17 contains the unit number from which to read the character. If the implementation does not support multiple terminal devices or if the devices are not directly addressable, R17 should be zero. The unit number from which the character was read is returned in R0<39:32>. If the implementation does not support multiple terminal devices, R0<39:32> is returned as zero.

GETC returns character reception status in R0<63:61>. If received characters are buffered by the console terminal, R0<61> is set to '1' whenever additional characters are available. If GETC returns a character without error, R0<63:62> is set to '00'. If no character is yet ready, R0<63:62> is set to '10'. If an error is encountered obtaining a character, R0<63:62> is set to '11'. Examples of errors during character reception include data overrun or loss of carrier.

When an error is returned by GETC, the contents of R0<31:0> and R0<60:48> depend on the capabilities of the underlying hardware. Implementations in which the hardware returns the character in error must provide that character in R0<31:0>. Additional device-specific error status may be contained in R0<60:48>.

When appropriate, GETC performs special keyboard operations such as turning keyboard LEDs on or off. Such action is based on the incoming stream of keycodes delivered by the console terminal.

The return address indicated by R26 should be mapped and executable by the kernel.

26.3.4.2 PUTS — Put Stream to Console Terminal

Format:

wcount = DISPATCH (PUTS,unit,address,length)

Inputs:

PUTS	= R16;	PUTS function code – 02 ₁₆
unit	= R17;	Terminal device unit number
address	= R18;	Virtual address of byte stream to be written
length	= R19;	Count of bytes to be written
retadr	= R26;	Return address

Outputs:

wcount	= R0;	Count of bytes written and status:
	R0<63:61>	‘000’ Success, all bytes written
		‘001’ Success, some bytes written
		‘100’ Failure, no bytes written Terminal error encountered
		‘111’ Failure, some bytes written Terminal error encountered
	R0<60:48>	Device-specific error status
	R0<47:32>	SBZ
	R0<31:0>	Count of bytes written (unsigned)

PUTS attempts to write a number of bytes to a console terminal device. R18 contains the base virtual address of the memory-resident byte stream; R19 contains its 32-bit size in bytes. The byte stream is written in order with no interpretation or special handling. The count of the bytes transmitted is returned in R0<31:0>.

Programming Note:

For multiple-byte character set encodings, the returned byte count may indicate a partial character transmission.

For implementations that support multiple terminal devices, R17 contains the unit number to which the byte stream is to be written; otherwise, R17 should be zero.

PUTS returns byte stream transmission status in R0<63:61>. If only a portion of the byte stream was written, R0<61> is set to ‘1’. If no error is encountered, R0<63:62> is set to ‘00’. If no bytes were written because the terminal was not ready, R0<63:62> is set to ‘10’. If an error is encountered writing a byte, R0<63:62> is set to ‘00’. Examples of errors during byte transmission include data overrun or loss of carrier.

When an error is returned by PUTS, additional device-specific error status may be contained in R0<60:48>.

Multiple invocations of PUTS may be necessary because the console terminal may accept only a very few bytes in a reasonable period of time.

The output byte stream located by R18 should be mapped and read accessible by the kernel; the return address indicated by R26 should be mapped and executable by the kernel.

26.3.4.3 RESET_TERM — Reset Console Terminal to Default Parameters

Format:

status = DISPATCH (RESET_TERM,unit)

Inputs:

RESET_TERM = R16; RESET_TERM function code – 03_{16}

unit = R17; Terminal device unit number

retadr = R26; Return address

Outputs:

status = R0; Status:

R0<63>	'0'	Success, terminal reset
	'1'	Failure, terminal not fully reset
R0<62:0>	SBZ	

RESET_TERM resets a console terminal device and its CTB to their initial, default state. All errors in the CTB are cleared. For implementations that support multiple terminal devices, R17 contains the unit number to be reset; otherwise, R17 should be zero.

The CTB describes the capabilities of the terminal device and its initial, default state. Depending on the terminal device type and particular console implementation, other terminal devices may be affected by the routine.

Programming Note:

For example, if multiple terminal units share a common interrupt, that interrupt may be disabled or enabled for all.

If the console terminal is successfully reset, RESET_TERM returns with R0<63> set to '0'. If errors are encountered, the routine attempts to return the console terminal to a usable state and then returns with R0<63> set to '1'.

The return address indicated by R26 should be mapped and executable by the kernel.

26.3.4.4 SET_TERM_INT — Set Console Terminal Interrupts

Format:

status = DISPATCH (SET_TERM_INT,unit,mask)

Inputs:

SET_TERM_INT = R16; SET_TERM_INT function code – 04₁₆
unit = R17; Terminal device unit number
mask = R18; Bit encoded mask:
R18<63:10> SBZ
R18<9:8> '01' No change to receive interrupts
'00' Disable receive interrupts
'1X' Enable receive interrupts
R18<7:2> SBZ
R18<1:0> '01' No change to transmit interrupts
'00' Disable transmit interrupts
'1X' Enable transmit interrupts

retadr = R26; Return address

Outputs:

status = R0; Status:
R0<63> '0' Success
'1' Failure, operation not supported
R0<62:2> SBZ
R0<0> '1' Transmit interrupts enabled
'0' Transmit interrupts disabled
R0<1> '1' Receive interrupts enabled
'0' Receive interrupts disabled

SET_TERM_INT reads, enables, and disables transmit and receive interrupts from a console terminal device and updates its CTB. For implementations that support multiple terminal devices, R17 contains the unit number to be reset; otherwise, R17 should be zero.

If the interrupt settings are successfully changed, the routine returns with R0<63> set to '0'. If the terminal device does not support the requested setting, the routine returns with R0<63> set to '1'.

Programming Note:

For example, a device that has a unified transmit/receive interrupt would not support a request to enable transmit interrupts while leaving receive interrupts disabled.

Regardless of success or failure, the routine always returns with the previous settings in R0<1:0>. The current state of the interrupt settings can be read without change by invoking SET_TERM_INT with R18<1:0> and R18<9:8> set to '01'.

The return address indicated by R26 should be mapped and executable by the kernel.

26.3.4.5 SET_TERM_CTL — Set Console Terminal Controls

Format

status = DISPATCH (SET_TERM_CTL, unit, ctb)

Inputs

SET_TERM_CTL = R16; SET_TERM_CTL function code – 05₁₆

unit = R17; Terminal device unit number

ctb = R18; Virtual address of CTB

retadr = R26; Return address

Outputs:

status = R0; Status:

R0<63>	'0'	Success, requested change completed
	'1'	Failure, change not completed
R0<62:32>	SBZ	
R0<31:0>		Offset to offending CTB field (unsigned)

SET_TERM_CTL, if successful, changes the characteristics of a console terminal device and updates its CTB. The changes are specified by fields contained in a CTB located by R18. The characteristics that can be changed, hence the active CTB fields, depend on the console terminal device type.

For implementations that support multiple terminal devices, R17 contains the unit number to be reset; otherwise, R17 should be zero.

If the console terminal characteristics are successfully changed, SET_TERM_CTL returns with R0<63> set to '0'. If errors are encountered or if the terminal device does not support the requested settings, the routine attempts to return the device to the previous usable state and then returns with R0<63> set to '1' and R0<31:0> set to the offset of an offending or unsupported field in the CTB located by R18. Regardless of success or failure, the device CTB table entry always contains the current device characteristics upon routine return. SET_TERM_CTL returns the CTB located by R18 without modification.

The CTB located by R18 should be mapped and read accessible by the kernel; the return address indicated by R26 should be mapped and executable by the kernel.

26.3.4.6 PROCESS_KEYCODE — Process and Translates Keycode

Format:

char = DISPATCH(PROCESS_KEYCODE,unit,keycode,again)

Inputs:

PROCESS_KEYCODE = R16; PROCESS_KEYCODE function code – 06₁₆
unit = R17; Terminal device unit number
keycode = R18; Keycode to be processed
again = R19; ‘1’ if calling again for same keycode
 ‘0’ otherwise
retadr = R26; Return address

Outputs:

char = R0; Translated character and status:
 R0<63:61> ‘000’ Success, character returned
 ‘101’ Failure, more time needed to
 process keycode
 ‘110’ Failure, device not supported
 by routine or routine not sup-
 ported
 ‘111’ Failure, no character; more
 keycodes needed or illegal
 sequence encountered
 R0<60> ‘0’ Success in correcting severe
 error
 ‘1’ Failure in correcting severe
 error
 R0<59:32> SBZ
 R0<31:0> Translated character

PROCESS_KEYCODE attempts to translate the keycode contained in R18 and, if successful, returns the character in R0<31:0>. The translation is based on the current character set encoding, language, and console terminal device state contained in the appropriate CTB. The translated character may be from one to four bytes. For implementations that support multiple terminal devices, R17 contains the unit number of the keyboard; otherwise, R17 should be zero.

Implementation Note:

For ISO Latin-1 character set encoding, PROCESS_KEYCODE returns a one-byte character.

PROCESS_KEYCODE returns keycode translation status in R0<63:61>. The processing falls into one of several cases:

1. The keycode, along with previous keycodes if any, translates into a character from the currently selected character set. In this case, R0<63:61> is set to '000'.
2. The keycode, along with previously entered keycodes if any, does not translate into a character from the currently selected character set. This is because either:
 - Not yet enough keycodes have been entered to produce a character in the currently selected character set.
 - The keycodes entered to this point indicate a severe keyboard error status.
 - The keycodes entered to this point form an illegal or unsupported keycode sequence.

In this case, R0<63:61> is set to '111'.

3. The console terminal device for which keycode translation is being performed is not supported by the PROCESS_KEYCODE implementation or the console implementation does not support PROCESS_KEYCODE. In this case, R0<63:61> is set to '110'.
4. The keycode cannot be processed in a reasonable amount of time; multiple invocations of PROCESS_KEYCODE are necessary. In this case, the routine returns with R0<63:61> set to '101'. The subsequent call(s) should be made with the same keycode in R18 and R19 set to '1'.

Implementation Note:

It may not be possible for an implementation to perform all the actions associated with special keycodes (such as turning on LEDs) in a timely manner. The PROCESS_KEYCODE routine must return after partial completion of an operation if necessary. It is the responsibility of the console to ensure that subsequent calls make forward progress. The delay between successive operating system calls is UNPREDICTABLE, although the operating system should attempt to complete the operation in a timely fashion. See Section 26.3.4.

In all but the first case, the contents of R0<31:0> are UNPREDICTABLE.

When certain severe keyboard errors are encountered, PROCESS_KEYCODE attempts to correct them by performing special keyboard operations. Those severe errors that may be corrected are device specific and contained in the terminal device CTB. If an error is encountered and the attempt to correct the error is unsuccessful, R0<60> is set to '1'; otherwise R0<60> is set to '0'.

The keyboard state recorded in the CTB is updated appropriately as the input stream of keycodes is processed. If appropriate, PROCESS_KEYBOARD may buffer some of the keycodes in the CTB keycode buffer. The supported keyboard state changes are device specific and are listed in the device CTB.

The return address indicated by R26 should be mapped and executable by the kernel.

26.3.4.7 CONSOLE_OPEN — Open Console Terminal

Format:

char = DISPATCH (OPEN_CONSOLE)

Inputs:

GETC = R16; OPEN_CONSOLE function code – 07₁₆
retadr = R26; Return address

Outputs:

char = R0; Returned character and status:
R0<63:61> '000' Success, console opened
 '100' Failure, console not opened
R0<60:48> Device-specific error status
R0<47:40> SBZ

CONSOLE_OPEN opens the console terminal input/output device for use. All other console terminal callbacks should be attempted only after a successful CONSOLE_OPEN.

CONSOLE_OPEN attempts to open the console terminal input/output device and, if successful, R0<63:61> is returned as '000', otherwise, R0<63:61> is returned as '100'. Additional device-specific error status may be contained in R0<60:48>.

The return address indicated by R26 should be mapped and executable by the kernel.

26.3.4.8 CONSOLE_CLOSE — Close Terminal

Format:

char = DISPATCH (CONSOLE_CLOSE)

Inputs:

CONSOLE_CLOSE = R16; CONSOLE_CLOSE function code – 08₁₆
retadr = R26; Return address

Outputs:

char = R0; Returned character and status:
R0<63:61> '000' Success, console closed
 '100' Failure, console not closed
R0<60:48> Device-specific error status
R0<47:0> SBZ

CONSOLE_CLOSE terminates use of the console terminal input/output device.

CONSOLE_CLOSE attempts to close the console terminal input/output device and, if successful, R0<63:61> is returned as '000', otherwise, R0<63:61> is returned as '100'. Additional device-specific error status may be contained in R0<60:48>.

The return address indicated by R26 should be mapped and executable by the kernel.

26.3.5 Console Generic I/O Device Routines

The Alpha console provides primitive generic I/O device routines for system software use during the bootstrap or crash process. These routines serve in place of the more sophisticated system software I/O drivers until such time as these drivers can be established. These routines may also be used to access console-private devices that are not directly accessible by the processor.

During the bootstrap process, these routines can be used to acquire a secondary bootstrap program from a system bootstrap device or write messages to a terminal other than the logical console terminal. When the operating system is about to crash, these routines can be used to write dump files.

These routines are *not* intended for use while the operating system is fully functional. These routines may:

- Alter the current IPL.

The console may raise the current IPL. It may lower the current IPL only insofar as the state presented to the operating system remains consistent, as though the IPL had not been lowered. The console must ensure that interrupts that would not have been delivered at the caller's IPL are pended and delivered to the operating system at the conclusion of the callback.

- Block interrupts.

These routines may cause any and all interrupts to be blocked or delivered to and serviced by the console for the duration of the routine execution.

- Block exceptions.

These routines may cause any and all exceptions to be blocked or delivered to and serviced by the console for the duration of the routine execution.

- Alter the existing memory management policy.

The console may substitute a console-private (or bootstrap address) mapping for the duration of the routine execution.

Programming Note:

The console must resolve any virtually addressed arguments before altering the existing memory management policy.

- Take any length of time for completion.

The operating system cannot guarantee timeliness when invoking these routines. Any operating system timer may have expired before their return. The time necessary for completion is UNPREDICTABLE; however, a console implementation will attempt to minimize the time whenever possible.

Before returning to the invoking system software, these routines must restore any altered processor state. These routines must return to the calling system software at the IPL and in the memory management policy of that software.

System software invokes these routines synchronously. When invoking these routines, system software must:

- Be executing in kernel mode.

If these routines are invoked in other modes, their execution causes UNPREDICTABLE operation.

- Be executing on the primary processor in a multiprocessor configuration.

If these routines are invoked on other processors, their execution causes UNDEFINED operation.

26.3.5.1 OPEN — Open Generic I/O Device for Access

Format:

channel = DISPATCH (OPEN,devstr,length)

Inputs:

OPEN = R16; OPEN function code – 10_{16}
devstr = R17; Starting virtual address of byte string that contains the device specification
length = R18; Length of byte buffer
retadr = R26; Return address

Outputs:

channel = R0; Assigned channel number and status:
R0<63:62> '00' Success
 '10' Failure, device does not exist
 '11' Failure, error, device cannot be
 accessed or prepared
R0<61:60> SBZ
R0<59:32> Device-specific error status
R0<31:0> Assigned channel number of device

OPEN prepares a generic I/O device for use by the READ and WRITE routines. R17 contains the base virtual address of a byte string that specifies the complete device specification of the I/O device. The length of the string is given in R18. The format and contents of the device specification string follow that of the BOOTED_DEV environment variable.

The routine assigns a unique channel number to the device. The channel number is returned in R0 and must be used to reference the device in subsequent calls to the READ, WRITE, and CLOSE routines.

OPEN returns status in R0<63:62>. If the I/O device exists and can be prepared for subsequent accesses, R0<63:62> is set to '00'. If the device does not exist, R0<63:62> is set to '10'. If the device exists, but errors are encountered in preparing the device, R0<63:62> is set to '11' and additional device-specific status is recorded in R0<59:32>. In the latter two failure cases, the channel number returned in R0<31:0> is UNPREDICTABLE.

All console implementations must support at least two concurrently opened generic I/O devices. Additional generic I/O devices may be supported.

For magnetic tape devices, OPEN does not affect the current tape position, nor is any rewind of the tape performed.

Multiple channels cannot be assigned to the same device; the second and any subsequent calls to OPEN fail with R0<63:62> set to '11' and R0<31:0> as UNPREDICTABLE. The status of the first opened channel is unaffected.

The input string located by R17 should be mapped and read accessible by the kernel; the return address indicated by R26 should be mapped and executable by the kernel.

26.3.5.2 CLOSE — Close Generic I/O Device for Access

Format:

status = DISPATCH (CLOSE,channel)

Inputs:

CLOSE = R16; CLOSE function code – 11₁₆

channel = R17; Channel to close

retadr = R26; Return address

Outputs:

status = R0; Status:

R0<63>	'0'	Success
	'1'	Failure
R0<62:60>	SBZ	
R0<59:32>	Device-specific error status	
R0<31:0>	SBZ	

CLOSE deassigns the channel number from a previously opened block storage I/O device. The channel number is free to be reassigned. The I/O device must be reopened before any subsequent accesses.

CLOSE returns status in R0<63>. If the channel was open and the close is successful, R0<63> is set to '0'; otherwise R0<63> is set to '1' and additional device-specific status is recorded in R0<59:32>.

For magnetic tape devices, CLOSE does not affect the current tape position, nor is any rewind of the tape performed.

The return address indicated by R26 should be mapped and executable by the kernel.

26.3.5.3 IOCTL — Perform Device-Specific Operations

Format:

count = DISPATCH (IOCTL,channel,R18,R19,R20,R21)

Inputs:

IOCTL = R16; IOCTL function code – 12_{16}
channel = R17; Channel number of device to be accessed
retadr = R26; Return address

For Magnetic Tape Devices Only:

operate = R18; Tape positioning operation:
 ‘01’ For skip to next/previous interrecord gap
 ‘02’ For skip over tape mark
 ‘03’ For rewind
 ‘04’ For write tape mark

count = R19; Number of skips to perform (signed)
 = R20 – Reserved for future use as inputs
 R21

Outputs:

For Magnetic Tape Devices Only:

count = R0; Number of skips performed and status:
 R0<63:62> ‘00’ Success
 ‘10’ Failure, position not found
 ‘11’ Hardware failure
 R0<61:60> SBZ
 R0<59:32> Device-specific error status
 R0<31:0> Number of skips actually performed (signed)

IOCTL performs special device-specific operations on I/O devices. The operation performed and the interpretation of any additional arguments passed in R18–R21 are functions of the device type as designated by the channel number passed in R17.

For magnetic tape devices, the following operations are defined:

- ‘01’ — IOCTL relocates the current tape position by skipping over a number of inter-record gaps. The direction of the skip and the number of gaps skipped is given by the signed 32-bit count in R19. Skipping with a count of ‘0’ does not change the current tape position. The number of gaps actually skipped is returned in R0<31:0>.

- ‘02’ — IOCTL relocates the current tape position by skipping over a number of tape marks. The direction of the skip and the number of marks skipped is given by the signed 32-bit count in R19. Skipping with a count of ‘0’ does not change the current tape position. The number of tape marks actually skipped is returned in R0<31:0>.
- ‘03’ — IOCTL rewinds the tape to the position just after the Beginning-of-Tape (BOT) marker. R0<31:0> is returned as SBZ.
- ‘04’ — IOCTL writes a tape mark starting at the current position. R0<31:0> is returned as SBZ.

IOCTL returns magnetic tape operation status in R0<63:62>. If the operation was successful, R0<63:62> is set to ‘00’. If the tape positioning was not successful, the tape is left at the position where the error occurred and R0<63:62> is set to ‘10’. Tape positioning may fail due to encountering a BOT marker (R18 ‘01’ or ‘02’), encountering a tape mark (R18 ‘01’), or running off the end of the tape. If a hardware device error is encountered, the final position of the tape is UNPREDICTABLE and R0<63:62> is set to ‘11’. In the event of an error, additional device-specific status is recorded in R0<59:32>.

The return address indicated by R26 should be mapped and executable by the kernel.

26.3.5.4 READ — Read Generic I/O Device

Format:

rcount = DISPATCH (READ,channel,count,address,block)

Inputs:

READ	= R16;	READ function code – 13_{16}
channel	= R17;	Channel number of device to be accessed
count	= R18;	Number of bytes to be read (should be multiple of the device's record length) (unsigned)
address	= R19;	Virtual address of buffer to read data into
block	= R20;	Logical block number of data to read (used only by disk devices)
retadr	= R26;	Return address

Outputs:

rcount	= R0;	Number of bytes read and status:
	R0<63>	'0' Success
		'1' Failure
	R0<62>	'1' EOT or Logical End of Device condition encountered
		'0' Otherwise
	R0<61>	'1' Illegal record length specified
		'0' Otherwise
	R0<60>	'1' Run off end of tape
		'0' Otherwise
	R0<59:32>	Device-specific error status
	R0<31:0>	Number of bytes actually read (unsigned)

READ causes data to be read from the generic I/O device designated by the channel number in R17 and written to a memory buffer pointed to by R19. The 32-bit transfer byte count, hence length of the buffer, is contained in R18. The buffer must be quadword aligned, virtually mapped, and resident in physical memory.

READ returns transfer status in R0<63:60> and the number of bytes actually read, if any, in R0<31:0>. If the routine is successful, R0<63> is set to '0'. If an error is encountered in accessing the device, R0<63> is set to '1'. Additional device-specific status may be returned in R0<59:32>.

The transfer byte count should be a multiple of the record length of the device. If the specified byte count is not a multiple of the record length, R0<61> is set to '1'. If the count exceeds the record length, the count is rounded down to the nearest multiple of the record length and

READ attempts to read that number of bytes. If the record length exceeds the count, it is UNPREDICTABLE whether READ attempts to access the device. If no read attempt is made, R0<63> is set to '1'.

For magnetic tape devices, READ does not interpret the tape format or differentiate between ANSI formatted and unformatted tapes. The routine reads the requested transfer byte count starting at the current tape position. READ terminates when one of the following occurs:

- The specified number of bytes have been read. In this case, R0<63:60> is set to '0000'.
- An interrecord gap is encountered. In this case, the tape is positioned to the next position after the gap and R0<63:60> is set to '0000'.
- A tape mark is encountered. In this case, the tape is positioned to the next position after the tape mark and R0<63:60> is set to '0100'. (After calling READ and finding a tape mark, the caller can determine if the logical End-of-Volume or an empty file section has been found by calling READ again. The condition exists if the second READ returns with zero bytes read and a tape mark found.)
- The routine runs off the end of tape. In this case, R0<63:60> is set to '1001'.

READ ignores End-of-Tape (EOT) markers.

For disk devices, READ does not understand the file structure of the device. The routine reads the requested transfer byte count starting at the logical block number specified by R20. The transfer continues until either the specified number of bytes has been read or the last logical block on the device has been read. If the logical end of the device is encountered, then R0<63:62> is set to '01'.

For network devices, READ interprets and removes any device-specific or protocol-specific packet headers. If a packet has been received, the remainder of the packet is copied into the specified buffer. If a packet has not been received, the routine returns with R0<31:0> set to '0'. Only those network packets that are specifically addressed to this system and are of the specified protocol type are returned; broadcast packets are not returned. The actual packet size is dependent on the device and protocol; the characteristics of the network device and protocol are specified at the time of the channel OPEN.

The buffer pointed to by R19 should be mapped and write accessible by the kernel; the return address indicated by R26 should be mapped and executable by the kernel.

26.3.5.5 WRITE — Write Generic I/O Device

Format:

wcount = DISPATCH (WRITE,channel,count,address,block)

Inputs:

WRITE	= R16;	WRITE function code – 14_{16}
channel	= R17;	Channel number of device to be accessed
count	= R18;	Number of bytes to be written (should be multiple of the device's record length) (unsigned)
address	= R19;	Virtual address of buffer to read data from
block	= R20;	Logical block number of data to be written (used only by disk devices)
retadr	= R26;	Return address

Outputs:

wcount	= R0;	Number of bytes written and status:
	R0<63>	'0' Success
		'1' Failure
	R0<62>	'1' EOT or Logical End of Device condition encountered
		'0' Otherwise
	R0<61>	'1' Illegal record length specified
		'0' Otherwise
	R0<60>	'1' If run off end of tape
		'0' Otherwise
	R0<59:32>	Device-specific error status
	R0<31:0>	Number of bytes actually written (unsigned)

WRITE causes data to be written to the generic I/O device designated by the channel number in R17 and read from a memory buffer pointed to by R19. The 32-bit transfer byte count, hence length of the buffer, is contained in R18. The buffer must be quadword aligned, virtually mapped, and resident in physical memory.

WRITE returns transfer status in R0<63:60> and the number of bytes actually written, if any, in R0<31:0>. If the routine is successful, R0<63> is set to '0'. If an error is encountered in accessing the device, R0<63> is set to '1'. Additional device-specific status may be returned in R0<59:32>.

The transfer byte count should be a multiple of the record length of the device. If the specified byte count is not a multiple of the record length, R0<61> is set to '1'. If the count exceeds the record length, the count is rounded down to the nearest multiple of the record length and

WRITE attempts to write that number of bytes. If the record length exceeds the count, it is UNPREDICTABLE whether WRITE attempts to access the device. If no write attempt is made, R0<63> is set to '1'.

For magnetic tape devices, WRITE does not interpret the tape format or differentiate between ANSI formatted and unformatted tapes. The routine writes the requested transfer byte count starting at the current tape position. WRITE terminates when any of the following occur:

- The specified number of bytes has been written without detecting an End-of-Tape (EOT) marker. In this case, R0<63:60> is set to '0000'.
- The specified number of bytes has been written and an End-of-Tape (EOT) marker was detected. In this case, R0<63:60> is set to '0100'.
- The routine runs off the end of tape. In this case, R0<63:60> is set to '1001'.

For disk devices, WRITE does not understand the file structure of the device. The routine writes the requested transfer byte count starting at the logical block number specified by R20. The transfer continues until either the specified number of bytes has been written or the last logical block on the device has been written. If the logical end of the device is encountered, then R0<63:62> is set to '01'.

For network devices, WRITE appends any device-specific or protocol-specific headers. The routine transmits the specified requested transfer bytes with the proper network protocol over the appropriate network. The actual packet size is dependent on the device and protocol; the characteristics of the network device and protocol are specified at the time of the channel OPEN.

The buffer pointed to by R19 should be mapped and write accessible by the kernel; the return address indicated by R26 should be mapped and executable by the kernel.

26.3.6 Console Environment Variable Routines

System software accesses the environment variables indirectly through console callback routines. These routines may be invoked while the operating system is fully functional as well as during operating system bootstrap or crash. The `GET_ENV`, `SET_ENV`, and `RESET_ENV` routines are subject to the constraints given in Section 26.3.1. These routines must:

- Not alter the current IPL or current mode.
These routines must be invoked in kernel mode.
- Not alter the existing memory management policy.
All internal pointers must be remapped by `FIXUP`.
- Not block interrupts.
The operating system must be capable of continuing to receive hardware and software interrupts.

The constraints on `SAVE_ENV` differ; see Section 26.3.6.4.

The time necessary for these routines to complete is `UNPREDICTABLE`; however, a console implementation will attempt to minimize the time whenever possible.

Software Note:

Implementations must limit the execution time of these routines to significantly less than the interval clock interrupt period.

The console implementation must ensure that any access to an environment variable is atomic. The console implementation must resolve multiple competing accesses by system software as well as competing accesses by system software and the console presentation layer.

When invoking these routines, system software must be executing in kernel mode. If these routines are invoked in other modes, their execution causes `UNPREDICTABLE` operation.

These routines may be invoked on both the primary and secondary processors in a multiprocessor configuration. It is recommended that system software serialize competing accesses to a given environment variable; a stale value may be returned if `GET_ENV` is invoked simultaneously with `SET_ENV` or `RESET_ENV`.

26.3.6.1 SET_ENV — Set an Environment Variable

Format:

status = DISPATCH (SET_ENV, ID, value, length)

Inputs:

SET_ENV	= R16;	SET_ENV function code - 20_{16}
ID	= R17;	ID of environment variable
value	= R18;	Starting virtual address of byte stream containing value
length	= R19;	Number of bytes in buffer (unsigned)
retadr	= R26;	Return address

Outputs:

status	= R0;	Status:
		R0<63:61> '000' Success
		'100' Failure, variable read-only
		'110' Failure, variable not recognized
		'111' Failure, byte stream exceeds value length
		R0<60:31> SBZ
		R0<31:0> Maximum value length (unsigned)

SET_ENV causes the environment variable specified by the ID in R17 to have the value specified by the byte stream value pointed to by the virtual address in R18. The size in bytes of the input buffer is contained in R19.

SET_ENV returns status in R0<63:61>. If the environment variable is successfully set to the new value, R0<63:61> is set to '000'. If the variable is not recognized, R0<63:61> is set to '110'. If the variable is read-only, the value is unchanged and R0<63:61> is set to '100'. If the input buffer exceeds the maximum value length, the value is unchanged and R0<63:61> is set to '111'. In all cases, the maximum value length is returned in R0<31:0>.

The byte stream indicated by R18 should be mapped and read accessible by the kernel; the return address indicated by R26 should be mapped and executable by the kernel.

26.3.6.2 RESET_ENV — Reset an Environment Variable

Format:

status = DISPATCH (RESET_ENV, ID, value, length)

Inputs:

RESET_ENV	= R16;	RESET_ENV function code – 21 ₁₆
ID	= R17;	ID of environment variable
value	= R18;	Starting virtual address of byte stream to contain returned value
length	= R19;	Number of bytes in buffer (unsigned)
retadr	= R26;	Return address

Outputs:

status	= R0;	Status:
		R0<63:61> '000' Success
		'001' Success, byte stream truncated
		'100' Failure, variable read-only
		'101' Failure, variable read-only, byte stream truncated
		'110' Failure, variable not recognized
		R0<60:32> SBZ
		R0<31:0> Count of bytes returned (unsigned)

RESET_ENV causes the environment variable specified by the ID in R17 to be reset to the system default value and that default value to be returned in the byte stream specified by the virtual address in R18. The size in bytes of the input buffer is contained in R19.

RESET_ENV returns status in R0<63:61>. If the environment variable is successfully reset to the default value, R0<63:62> is set to '00'. If the variable is recognized but read-only, the value is unchanged and R0<63:62> is set to '10'. In both cases, the default value is copied into the byte stream and R0<31:0> is set to the number of bytes copied; if the value must be truncated, R0<61> is set to '1'. If the variable is not recognized, R0<63:61> is set to '110' and R0<31:0> is set to '0'.

The byte stream indicated by R18 should be mapped and write accessible by the kernel; the return address indicated by R26 should be mapped and executable by the kernel.

26.3.6.3 GET_ENV — Get an Environment Variable

Format:

status = DISPATCH (GET_ENV, ID, value, length)

Inputs:

GET_ENV = R16; GET_ENV function code – 22_{16}
ID = R17; ID of environment variable
value = R18; Starting virtual address of buffer to contain returned value
length = R19; Number of bytes in buffer (unsigned)
retadr = R26; Return address

Outputs:

status = R0; Status:
R0<63:61> '000' Success
'001' Success, byte stream truncated
'110' Failure, variable not recognized
R0<60:32> SBZ
R0<31:0> Count of bytes returned (unsigned)

GET_ENV causes the value of the environment variable specified by the ID in R17 to be returned in the byte stream specified by the virtual address in R18. The size in bytes of the input buffer is contained in R19.

GET_ENV returns status in R0<63:61>. If the environment variable is recognized, R0<63:62> is set to '00', its current value is copied into the byte stream, and R0<31:0> is set to the number of bytes copied. If the value must be truncated, R0<61> is set to '1'. If the variable is not recognized, R0<63:61> is set to '110' and R0<31:0> is set to '0'.

The byte stream indicated by R18 should be mapped and write accessible by the kernel; the return address indicated by R26 should be mapped and executable by the kernel.

26.3.6.4 SAVE_ENV — Save Current Environment Variables

Format:

status = DISPATCH (SAVE_ENV)

Inputs:

SAVE_ENV = R16; SAVE_ENV function code – 23₁₆

retadr = R26; Return address

Outputs:

status = R0; Status:

R0<63:61>	'000'	Success, all values saved
	'001'	Success, some bytes saved, additional values to be saved
	'110'	Failure, routine unsupported
	'111'	Failure, error encountered saving values
R0<60:0>	SBZ	

SAVE_ENV attempts to update the nonvolatile storage of those environment variables that must be retained across console initializations and system power transitions.

Programming Note:

For example, SAVE_ENV may cause an EEPROM to be updated. That update may write all "NV" environment variable values to the EEPROM, or may only write those variables that have been modified since the last update or console initialization.

This routine is not subject to the constraints given in Section 26.3.6. The console may usurp operating system control of the system platform hardware, but must restore any such control or altered state before return. The console must not service any interrupts or exceptions that are otherwise intended for the operating system.

The nonvolatile storage update may take significant time and multiple invocations of SAVE_ENV may be necessary. The time necessary for this routine to complete is UNPREDICTABLE. A console implementation will attempt to minimize the time whenever possible and must return in a timely fashion. The routine must return after partial operation completion if necessary. It is the responsibility of the console to ensure that subsequent calls make forward progress. The operating system may delay for extended periods between subsequent calls; the console must not rely on timely invocations of SAVE_ENV.

Implementation Note:

Implementations must limit the execution time of these routines to significantly less than the interval clock interrupt period. A return after partial operation completion is preferable to long latency.

SAVE_ENV returns status on the update in R0<63:61>. When the update has successfully completed and all relevant variables have been saved, the routine returns with R0<63:61> set to '000'. If SAVE_ENV returns after only a partial update to ensure timely response, R0<63:61> is set to '001'. If an unrecoverable error is encountered, the routine returns with R0<63:61> set to '111'. The contents of the nonvolatile storage are UNDEFINED.

Implementation of SAVE_ENV is optional. If the console does not support SAVE_ENV, the routine returns with R0<63:61> set to '110'.

On a multiprocessor system with an embedded console, the routine must be invoked on each processor in the configuration. Section 27.8.1

It is recommended that system software ensure that calls to SET_ENV or RESET_ENV are not issued while an update operation is in progress on any processor. It is UNPREDICTABLE whether the updated environment value is saved.

The return address indicated by R26 should be mapped and executable by the kernel. This routine does not affect the current value of any environment variable maintained by the console.

26.3.7 Miscellaneous Routines

26.3.7.1 PSWITCH — Switch Primary Processors

Format:

status = DISPATCH (PSWITCH,action)

Inputs:

PSWITCH = R16; PSWITCH function code – 30_{16}
action = R17; Action requests:
R17<63:2> SBZ
R17<1:0> '01' Transition from primary
'10' Transition to primary
'11' Switch primary

cpu_id = R18; New primary CPU ID
retadr = R26; Return address

Outputs:

status = R0; Status:
R0<63> '0' Success
'1' Failure, operation not supported
R0<62:0> Implementation-specific error status

PSWITCH attempts to perform any implementation-specific functions necessary to support primary switching. R17 indicates the requested primary transition action. R18 contains the CPU ID (WHAMI IPR) of the new primary.

PSWITCH is invoked by the old primary, the secondary that is to become the new primary, or both. See Section 27.5.6 for a full description of PSWITCH usage, functionality, and error returns.

If PSWITCH is successful, it returns with R0<63> set to '0'. If PSWITCH is unsuccessful for any reason, it returns with R0<63> set to '1' and implementation-specific status in R0<62:0>.

PSWITCH is invoked at the highest IPL level or it produces UNDEFINED results. The return address indicated by R26 should be mapped and executable by the kernel.

26.3.7.2 FIXUP — Fixup Virtual Addresses in Console Routines

Format:

status = FIXUP (NEW_BASE_VA, HWRPB_VA)

Inputs:

NEW_BASE_VA = R16; New starting virtual address of the console callback routines
HWRPB_VA = R17; New starting virtual address of the HWRPB
retadr = R26; Return address

Outputs:

status = R0; Status:
R0<63> '0' Success
 '1' Failure
R0<62:0> SBZ

FIXUP adjusts virtual address references in all other console callback routines using the new starting virtual address in R16, the new starting virtual address of the HWRPB in R17, and the current contents of the CRB. See Section 26.3.8.1.2 for a full description of FIXUP usage and functionality.

If FIXUP is successful, it returns with R0<63> set to '0'. If FIXUP is not successful, console internal state has been compromised. The console attempts a cold bootstrap if the state transition in Figure 27–1 indicates a bootstrap and the BOOT_RESET environment variable is set to "ON" (4E4F₁₆). Otherwise, the system remains in console I/O mode.

This routine must be called in kernel mode and in the context of the existing memory mapping; otherwise its execution causes UNPREDICTABLE or UNDEFINED operation.

Software Note:

FIXUP must be called while the original address space mapping is in effect.

The return address indicated by R26 should be mapped and executable by the kernel.

26.3.7.3 BIOS_EMUL — Run BIOS Emulation Callback

Format:

status = DISPATCH (BIOS_EMUL, int86, input_flags, x86_regs, additional_data)

Inputs

func_code	= R16;	BIOS_EMUL function code – 32_{16}
int86	= R17;	BIOS interrupt number (also called the BIOS service number)
input_flags	= R18;	The following input flags: R18<63:5> SBZ R18<4> '1' Use data in R20 '0' Ignore R20 R18<3:1> Type of BIOS emulator to service the call: '000' 16-bit emulator type '001' 32-bit emulator type '010' 64-bit emulator type '011' Reserved '100' Reserved '101' Reserved '110' Reserved '111' Reserved R18<0> Type of call: '1' Emulator type inquiry '0' Service
x86_regs	= R19;	Virtual address of x86 register data block that represents the x86 register set for BIOS calls. Use the appropriate register structure for the type of BIOS emulator: 16-bit emulator — Use register structure 1 (Figure 26–4) 32-bit emulator — Use register structure 2 (Figure 26–4) 64-bit emulator — Not defined for this version of the architecture
Additional_data	= R20;	Virtual address of additional argument data. Specific to BIOS call
Retaddr	= R26;	Return address

Outputs:

status = R0 Status:

If R18<0> = 0, R0 has the following meaning:

R0<63>	'0'	Callback supported
	'1'	Callback not supported
R0<62>	'0'	Emulator type supported
	'1'	Emulator type not supported
R0<61>	'0'	Service number supported
	'1'	Service number not supported
R0<60:56>	SBZ	
R0<55:0>	Implementation-specific	

If R18<0> = 1, R0 has the following meaning:

R0<63>	'0'	Callback supported
	'1'	Callback not supported
R0<62:59>	SBZ	
R0<58:56>	Return console's emulator type:	
	'000'	No emulator in this console
	'001'	16-bit emulator in this console
	'010'	32-bit emulator in this console
	'011'	64-bit emulator in this console
R0<55:0>	SBZ	

The resulting x86 register state from the BIOS call is placed in the data block located at x86_regs (R19). Success or failure of the BIOS call is specific to the attempted call and the expected result in x86_regs.

The BIOS_EMUL callback provides access to the BIOS emulator, allowing emulation of the x86 INT assembler instruction.

The int86 value specifies the BIOS interrupt number to be emulated. A data block representing the x86 register set is used as input and is updated on return because operation of BIOS calls requires setting the x86 register set before the BIOS call and receiving data in them as the result of a BIOS call.

Programming Notes:

If a platform or pre-existing version of the firmware does not support BIOS_EMUL, R0<63> returns '1'.

The caller can determine the type of BIOS emulator in the console by setting R18<0> to '1'. BIOS_EMUL returns the type in R0<58:56>.

Because multiple BIOS emulators can be built into the console, use R18<3:1> to specify the type of BIOS emulator and register structure. If the console does not support a specified type, R0<63> and R0<62> return '1'.

BIOS_EMUL supports only INT10 service calls, and for any other service number, R0<63> returns '1'.

The caller should maintain the integrity of the register structure as input/output across multiple calls. The routine uses the register structure values as passed and returns the end values in the same structure.

The return address indicated by R26 should be mapped and kernel-executable.

Figure 26–4: BIOS Emulator Register Structures

Register Structure 1				Register Structure 2				
31	24	23	0	31	24	23	0	
SBZ	SBZ	AH	AL	EAX				VA
SBZ	SBZ	BH	BL	EBX				+4
SBZ	SBZ	CH	CL	ECX				+8
SBZ	SBZ	DH	DL	EDX				+12
SBZ	SBZ	SP		ESP				+16
SBZ	SBZ	BP		EBP				+20
SBZ	SBZ	SI		ESI				+24
SBZ	SBZ	DI		EDI				+28
SBZ	SBZ	CS		SBZ	SBZ	CS		+32
SBZ	SBZ	DS		SBZ	SBZ	DS		+36
SBZ	SBZ	SS		SBZ	SBZ	SS		+40
SBZ	SBZ	ES		SBZ	SBZ	ES		+44
SBZ	SBZ	FLAGS		EFLAGS				+48
SBZ	SBZ	IP		EIP				+52

Background Notes on BIOS Emulation:

- BIOS

BIOS, or Basic Input Output System, is firmware that initializes the hardware and sets it to a known state or to a state that is chosen by the hardware vendor or the system user. The BIOS code performs a power-up self-test (POST), configures buses and devices, and provides an interface to boot the operating system. BIOS code can also provide a set of functions that allows other system software to program devices to a given mode or state. Those functions are device-dependent, but follow an industry standard that is supported by most hardware vendors. Most BIOS code is written in the x86 assembly language.

- BIOS Emulation

To support standard BIOS firmware (x86-based) on Alpha-based platforms, the Alpha console has a built-in emulator that emulates the x86 instruction set. The emulator supports VGA BIOS functions and is limited to the less complex, INT10, VGA BIOS calls. The emulator supports a large number of third-party graphics cards on

Alpha-based platforms.

The emulator can be 16 bit or 32 bit. A 16-bit emulator limits its support to the 16-bit register and instruction sets. A 32-bit emulator supports the 32-bit register and instruction sets, as well as the 16-bit instruction set.

- BIOS_EMUL Callback Routine

The BIOS_EMUL callback routine provides a generic interface to the BIOS emulator. It provides a mechanism to request the console's BIOS emulator type and returns appropriate status and error codes that indicate supported and unsupported arguments. Operating systems require this interface to support third-party graphics cards for different Alpha platforms.

Commodity PC graphics cards (SVGA) rely heavily on the BIOS to set the graphics mode. Vendors generally do not document how to set a graphics mode by register programming (like 1280x1024), but instead refer to the BIOS INT10 call, which is used to set up the card. Without the interface provided by BIOS_EMUL, the operating system has no access to BIOS emulation, and the graphics cards must be programmed by specialized code in the driver. Further, BIOS_EMUL allows the operating system to maintain support for graphics cards when vendors release new versions, because the interface lets the operating system continue to correctly interact with any changed mode parameters.

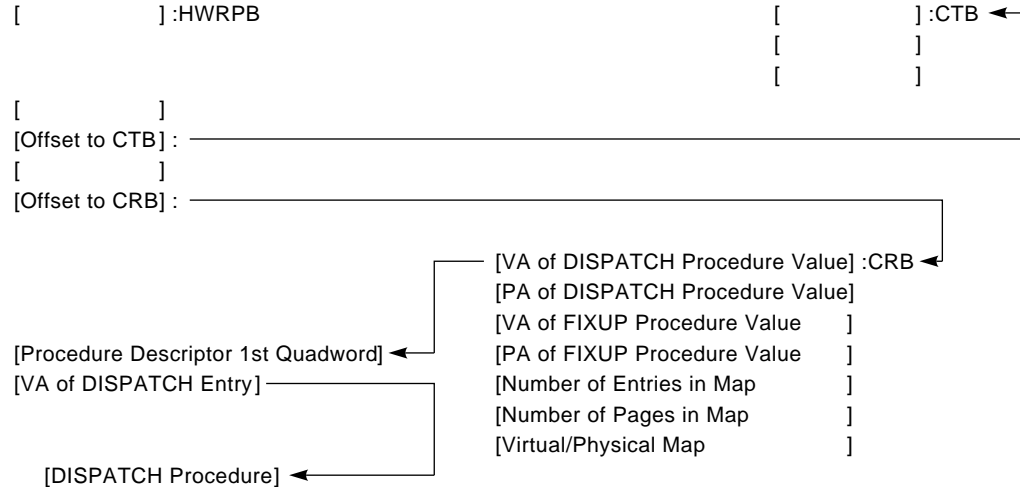
26.3.8 Console Callback Routine Data Structures

The console and system software share two data structures that are necessary for the console callback routines: the Console Routine Block (CRB) and the Console Terminal Block (CTB) table. Both are located by offset fields in the HWRPB as shown in Figure 26–5.

The CRB locates all addresses necessary for console callback routine function. The base physical address of the CRB is obtained by adding the CRB OFFSET field at HWRPB[192] to the base physical address of the HWRPB. The CRB format is shown in Figure 26–6 and described in Table 26–10.

The CTB table contains information necessary to describe the console terminal devices. The base physical address of the CTB table is obtained by adding the CTB TABLE OFFSET field at HWRPB[184] to the base physical address of the HWRPB. The CTB format is shown in Figure 26–7 and described in Table 26–8.

Figure 26–5: Console Data Structure Linkage



26.3.8.1 Console Routine Block

Before transferring control to system software, the console ensures that the console callback routines, console-private data structures, and associated local I/O space locations are mapped into region 0 of initial bootstrap address space. All necessary pages are located by the console routine block (CRB).

Figure 26–6 Console Routine Block

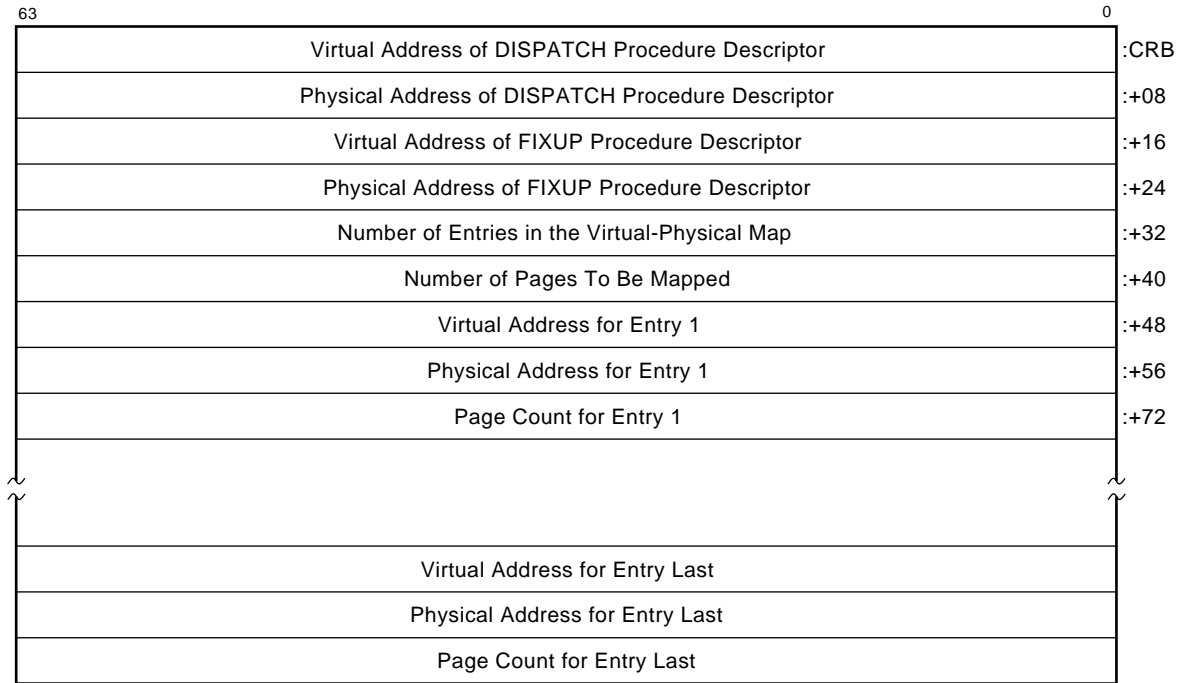


Table 26–10 CRB Fields

Offset	Description												
CRB	DISPATCH VA — The virtual address of the OpenVMS procedure descriptor for the DISPATCH procedure.												
+08	DISPATCH PA — The physical address of the OpenVMS procedure descriptor for the DISPATCH procedure.												
+16	FIXUP VA — The virtual address of the OpenVMS procedure descriptor for the FIXUP procedure.												
+24	FIXUP PA — The physical address of the OpenVMS procedure descriptor for the FIXUP procedure.												
+32	ENTRIES — The number of entries in the virtual-physical map. Unsigned integer.												
+40	PAGES — The total number of physical pages to be mapped. Unsigned integer.												
+48	ENTRY — Each entry identifies a collection of physically contiguous pages to be mapped. Each map entry consists of three quadwords:												
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Offset</th> <th style="text-align: left;">Name</th> <th style="text-align: left;">Description</th> </tr> </thead> <tbody> <tr> <td>+00</td> <td>ENTRY_VA</td> <td>Base virtual address for entry</td> </tr> <tr> <td>+08</td> <td>ENTRY_PA</td> <td>Base physical address for entry</td> </tr> <tr> <td>+16</td> <td>ENTRY_PAGES</td> <td>Number of contiguous physical pages to be mapped. Unsigned integer.</td> </tr> </tbody> </table>	Offset	Name	Description	+00	ENTRY_VA	Base virtual address for entry	+08	ENTRY_PA	Base physical address for entry	+16	ENTRY_PAGES	Number of contiguous physical pages to be mapped. Unsigned integer.
Offset	Name	Description											
+00	ENTRY_VA	Base virtual address for entry											
+08	ENTRY_PA	Base physical address for entry											
+16	ENTRY_PAGES	Number of contiguous physical pages to be mapped. Unsigned integer.											

The CRB must be quadword aligned. The DISPATCH and FIXUP addresses must be quadword aligned; all unused bits should be zero. The ENTRY addresses must be page aligned and all unused bits should be zero.

The DISPATCH and FIXUP procedure descriptors located by DISPATCH_PA, DISPATCH_VA, FIXUP_PA and FIXUP_VA must be contained within the pages located by the first virtual-physical map entry.

26.3.8.1.1 Console Routine Block Initialization

Before transferring control to system software, the console initializes all fields of the CRB. The console fills in all physical and virtual address fields, the number of entries in the virtual-physical map (ENTRIES), the total number of pages to be mapped (PAGES), and the virtual addresses contained in the OpenVMS procedure descriptors for the DISPATCH and FIXUP procedures.¹ PAGES is the sum of the contents of all ENTRY_PAGES fields.

All addresses are initially mapped within region 0 of the initial bootstrap address space. These addresses include the contents of the CRB and all addresses contained within the DISPATCH and FIXUP procedure descriptors. The mapping must permit kernel access with appropriate read/write/execute access. The KRE, KWE, and FOx PTE fields are never subsequently altered by system software. The initial mapping need not be virtually contiguous.

26.3.8.1.2 Console Routine Remapping

When the console transfers control to the system software, the console callback routines may be invoked by the system software without additional setup. All necessary virtual mappings into initial bootstrap address space must be performed by the console before transferring control.

The system software may virtually remap the console callback routines. Remapping permits the system software to relocate the routines to virtual addresses other than those assigned in initial bootstrap address space. Relocation requires that the console adjust (or fix up) various internal virtual address references.

The system software invokes the FIXUP routine to enable the console to perform the necessary internal relocations. The FIXUP routine virtually relocates all console routines and adjusts any console-private virtual address pointers such as those used to locate a local I/O device or HWRPB data structure. If system software virtually remaps the HWRPB, FIXUP must be invoked before calling any other console callback routine; it is recommended that system software remap both the HWRPB and the console routines together. Calling the console callback routines after the HWRPB has been remapped from its original bootstrap address location results in UNDEFINED operation of the system.

To remap the console callback routines, the system software and the console cooperate as follows:

1. System software must be executing on the primary processor in a multiprocessor system.
2. System software determines the new base virtual address of the HWRPB; this remapping is optional. System software does not perform any remapping of the HWRPB at this step.

¹ The OpenVMS calling standard specifies that the second quadword of a procedure descriptor contains the entry address (virtual) of the procedure itself.

System software need not remap the memory data descriptor table located by HWRPB[200]. See Section 26.1 for a description of the HWRPB and its size.

3. System software determines the new base virtual address of the console callback routines. The CRB entries will be mapped into a set of virtually contiguous pages. The CRB PAGES field (CRB[40]) is used to determine the number of pages that must be mapped. System software does not perform any remapping of the console callback routines at this step.
4. System software passes control to the console by calling FIXUP (NEW_BASE_VA, NEW_HWRPB_VA), initiating the remapping. NEW_BASE_VA is the new base virtual address as established in step 3. NEW_HWRPB_VA is the new starting virtual address of the HWRPB as established in step 2. The remapping process is only initiated at this step; do not attempt to access the HWRPB or CRB using the new VAs.
5. The console first locates the HWRPB, then locates the CRB using the CRB OFFSET field. The console then locates all internal pointers and adjusts them. All linkage sections and other console-internal pointers must be modified. These data structures can be located during FIXUP because the initial bootstrap address space mapping is in effect; any console-internal pointers are valid until modified.

System software need not remap the optional CONFIG block or FRU table located by HWRPB OFFSET fields. If these blocks are physically contiguous to the HWRPB and the required offset blocks and if they will subsequently be used by the console, they must be located by console-internal pointers and those pointers must be modified during FIXUP.

DISPATCH and FIXUP are not uniquely remapped by the system software. The FIXUP must update the DISPATCH and FIXUP procedure descriptors located by CRB[8] and CRB[24]. The physical pages containing the procedure descriptors and the routines themselves must be included in the virtual-physical map.

The relative virtual address offsets of the pages located by the entry map are not guaranteed to be retained across the FIXUP. The initial bootstrap address mapping of the physical pages located by the entry map is not required to be virtually contiguous. The system software remapping is required to be virtually contiguous. Any offsets that cross physical pages may have to be modified by FIXUP.

6. The console returns from FIXUP. If the FIXUP was not successful, console internal state has been compromised. The console attempts a cold bootstrap if the state transition in Figure 27-1 indicates a bootstrap and the BOOT_RESET environment variable is set to "ON" (4E4F₁₆). Otherwise, the system remains in console I/O mode.
7. System software updates each virtual-physical map entry of the CRB:
 - a. The PTE and TB entries that correspond to the range of old virtual address are invalidated using the old ENTRY_VA and ENTRY_PAGES values.
 - b. The new starting virtual address is written into the ENTRY_VA. This virtual address is computed by adding the NEW_BASE_VA to the sum of the PAGE_COUNTs of each preceding entry.
 - c. New PTEs are constructed for each physical page. The new PTE FOx and protection fields are copied from the original bootstrap address PTE.

Programming Note:

It is the responsibility of the console to judiciously set both the protection and FOx

bits in the bootstrap address PTE. In particular, if the console sets the FOE bit, there is no architectural guarantee that the console exception handler will gain control as a result, nor is there any obvious appropriate response for the operating system handler.

8. System software updates the DISPATCH and FIXUP VA's. The first virtual-physical map entry locates the physical page that contains the DISPATCH and FIXUP procedure descriptors.
9. System software updates all PTEs and invalidates all appropriate TB entries associated with the remapped HWRPB and any remapped OFFSET blocks.

At the completion of this process, the console callback routines are remapped and may again be used by system software. Since FIXUP itself is relocated, system software may remap the routines more than once.

26.3.8.2 Console Terminal Block Table

The Console Terminal Block (CTB) table indicates the current identity and characteristics of each console terminal device. The CTB table is the only data structure shared by the console and system software that describes the terminal devices accessible by console callback routines.

The CTB table contains an array of CTBs. Each CTB is a quadword-aligned structure with format as shown in Figure 26-7 and described in Table 26-8. The index of the CTB in the CTB table is the unit number of the terminal device. The CTB format consists of two parts: a header and a device-specific segment. The format of the header is common to all CTBs; the format of the device-specific segment is dependent on the unique device type.

There is *only one* console terminal. The console terminal unit is selected by the console presentation layer before bootstrapping the operating system. See Section 25.3. Once the operating system is bootstrapped, the console terminal unit should not be changed by the console presentation layer. Any attempt to do so results in UNDEFINED operation of the console. Specifically, if the console presentation layer halts the operating system, alters the console terminal unit, then restarts or continues operating system execution, the operation of the console is UNDEFINED. The console terminal unit is identified by the TTY_DEV environment variable. During console initialization, the console:

1. Locates all console terminal devices.
2. Selects the console terminal.
3. Builds a CTB for each.
4. Initializes the CTB OFFSET field of the HWRPB.
5. Initializes each console terminal device.
6. Records the default state of each console terminal device in its CTB.
7. Records the unit number of the console terminal in the TTY_DEV environment variable.

Whenever the console changes the state of a console terminal device, the console must update its CTB to reflect the change. The console may record extended status on character transfers (GETC/PUTS) in the CTB.

System software uses the CTB to determine console terminal device characteristics. System software never directly modifies the contents of a CTB; such modifications can result in UNDEFINED operation of the console terminal device either as the result of a subsequent call to a console terminal routine or as the result of a console internal need to access a console terminal device (for example, as the result of a halt). System software calls the SET_TERM_CTL console terminal routine to change console terminal device characteristics.

Figure 26–7 Console Terminal Block

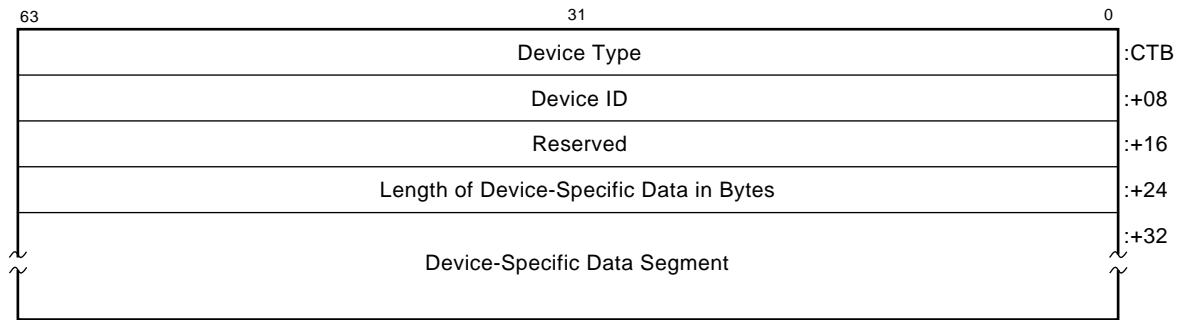


Figure 26–8 CTB Fields

Offset	Description														
CTB	<p>DEVICE TYPE — Console terminal device type and format of the device-specific segment. Defined device types are:</p> <table border="1"> <thead> <tr> <th>Type</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No console present</td> </tr> <tr> <td>1</td> <td>Detached service processor</td> </tr> <tr> <td>2</td> <td>Serial line UART</td> </tr> <tr> <td>3</td> <td>Graphics display with LK keyboard connected to serial line UART</td> </tr> <tr> <td>4</td> <td>Multipurpose</td> </tr> <tr> <td>Other</td> <td>Reserved</td> </tr> </tbody> </table>	Type	Description	0	No console present	1	Detached service processor	2	Serial line UART	3	Graphics display with LK keyboard connected to serial line UART	4	Multipurpose	Other	Reserved
Type	Description														
0	No console present														
1	Detached service processor														
2	Serial line UART														
3	Graphics display with LK keyboard connected to serial line UART														
4	Multipurpose														
Other	Reserved														
+08	<p>DEVICE ID — The physical device and channel that sends and receives the console terminal stream. This field is necessary for configurations that include multiple-channel devices or multiple single-channel devices. The field has two subfields:</p> <table border="1"> <thead> <tr> <th>Bits</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><63:32></td> <td>Device index</td> </tr> <tr> <td><31:0></td> <td>Channel index</td> </tr> </tbody> </table> <p>For implementations that support only a single directly connected console terminal device, this field is set to zero. The device ID is not necessarily related to the console terminal device unit number.</p>	Bits	Description	<63:32>	Device index	<31:0>	Channel index								
Bits	Description														
<63:32>	Device index														
<31:0>	Channel index														

Figure 26–8 CTB Fields (Continued)

Offset	Description
+16	RESERVED — This field is reserved for future expansion and may not be used by the console or system software.
+24	DSD LENGTH — This field specifies the number of bytes in the device-specific data field, DSD.
+32	DSD — This field contains device-specific data associated with the unique console terminal type. Device-specific data may include such parameters as baud rate, flow control enable, and the current state of the CAPS LOCK key. The DSD field should contain only those items that must be shared between the console and system software.

26.4 Interprocessor Console Communications

This section considers only those communications between a running processor and a console processor. Communications paths between running processors are external to the console. Communications paths between console processors are internal to the console.

Commands are transmitted from a running primary to a console secondary; messages (and requests) are transmitted from a console secondary to a running primary. Messages and requests may also be passed from the console primary to the running primary. This can occur when the primary processor is temporarily in console mode and wants to pass an unsolicited message to the operating system before returning to program mode. The message passing mechanism is identical to that used by console secondaries.

Commands and messages are passed via receive (RX) and transmit (TX) buffers contained in each per-CPU slot of the HWRPB. The use of these buffers is controlled by the Receive Buffer Ready (RXRDY) and Transmit Buffer Ready (TXRDY) flags.

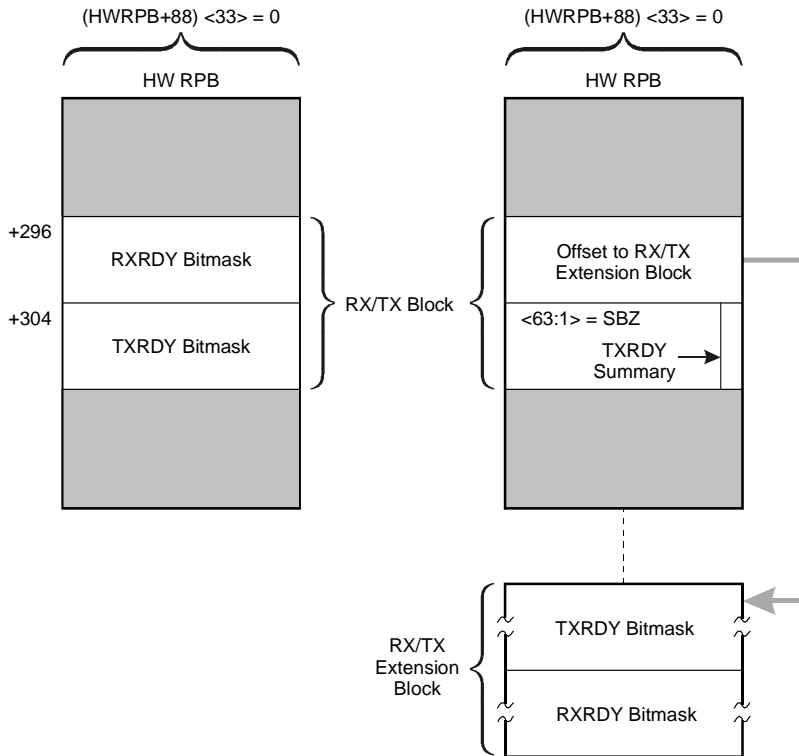
The transmit and receive buffers are named from the point of view of the console secondary. The console secondary receives commands in the RX buffer and transmits messages in the TX buffer.

26.4.1 Interprocessor Console Communications Flags

The Receive Buffer Ready (RXRDY) and Transmit Buffer Ready (TXRDY) flags are used to control the interprocessor console communications. The RXRDY and TXRDY flags are gathered into bitmasks in the HWRPB at one of two possible locations, determined by the RX/TX EXTENT bit <33> in the System Variation Field (HWRPB+88), as shown in Table 26–2.

The mapping of the RXRDY and TXRDY flags, as determined by the RX/TX EXTENT bit, is shown in Figure 26–9.

Figure 26–9: RXRDY and TXRDY Bitmasks in the HWRPB



As shown in Figure 26–9, if the RX/TX EXTENT bit (<33>) in the System Variation Field is clear, then the RX/TX Block in the HWRPB directly contains the RXRDY and TXRDY bitmasks. Each bitmask is exactly 64 bits in length, constraining the CPU namespace to be from 0 to 63.

If the RX/TX EXTENT bit is set, the RX/TX Block in the HWRPB does not contain the RXRDY and TXRDY bitmasks. Instead, the first quadword in the RX/TX Block contains an offset from the start of the HWRPB to an RX/TX Extension Block, where the RXRDY and TXRDY bitmasks may be found. The length of each bitmask is a function of the value stored at HWRPB+144, which is rounded up to the nearest multiple of 64, then divided by 64 to determine a quadword count for each bitmask. The bitmasks must each contain at least one quadword.

Implementation Note:

HWRPB revision #11 introduces the extended RX/TX mechanism. Existing software coded to earlier HWRPB revisions is not required to examine the RX/TX Extent bit in the System Variation Field. Firmware updates for existing platforms are not allowed to implement this extension without coordinating with software.

New platforms are encouraged to implement only the extended RX/TX mechanism, regardless of CPU count. This allows platform specific software to avoid having to examine the state of the RX/TX Extent bit.

The running primary sets the appropriate RXRDY flag to indicate to the receiving console secondary that a command is contained in the secondary's RX buffer. The secondary is assumed to be polling its RXRDY flag. The RXRDY flag is cleared by the secondary after the command has been read from the RX buffer and before executing the command.

A console secondary sets its TXRDY flag to indicate to the running primary that a message is contained in the secondary's TX buffer. The console generates an interprocessor interrupt to the primary to notify it that a message is ready. System software clears the TXRDY flag after the message has been read from the TX buffer and before processing the message.

Implementation Note:

The quadword at HWRPB+304 minimizes interprocessor interrupt service overhead by reducing the number of required memory lookups.

26.4.2 Interprocessor Console Communications Buffer Area

Each per-CPU slot of the HWRPB includes an RXTX Buffer Area that provides the communications path between processors. The buffer area is controlled by the RXRDY and TXRDY flags. The format is shown in Figure 26–10 and described in Table 26–11.

Figure 26–10 Inter-Console Communications Buffer

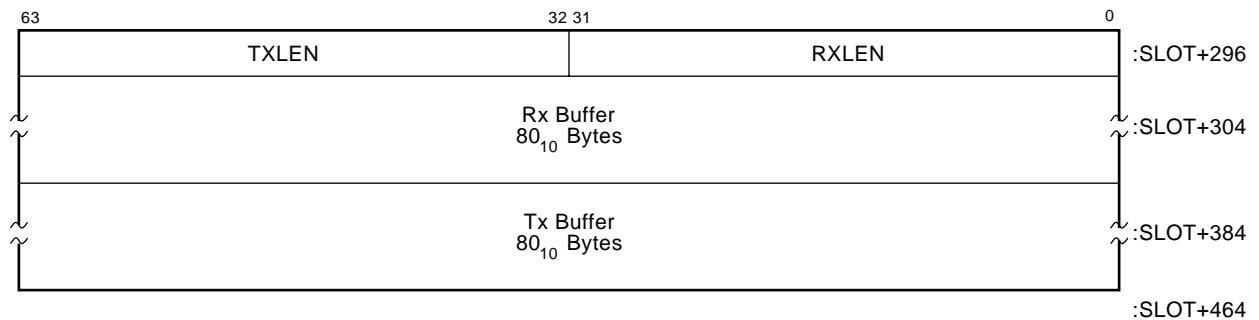


Table 26–11 Inter-Console Communications Buffer Fields

Offset	Description
SLOT+296	RXLEN — If the bit corresponding to this processor is set in the RXRDY bitmask, the RXLEN field contains the length in bytes of the command in the RX buffer.
+300	TXLEN — If the bit corresponding to this processor is set in the TXRDY bitmask, the TXLEN field contains the length in bytes of the message in the TX buffer.
+304	RX BUFFER — Buffer used by this console secondary to receive a command from the running primary. Only command data is passed through this buffer; a console secondary does not receive messages from the running primary. Commands must end with "<CR><LF>" (0A0D ₁₆).
+384	TX BUFFER — Buffer used by this console secondary (or primary temporarily in console mode) to transmit a message to the running primary. Only message data is passed through this buffer; a console secondary (or primary temporarily in console mode) does not send commands to the running primary. Messages must end with the console secondary's prompt, "<CR><LF>Pnn>>>" (3E3E 3E nn 50 0A0D ₁₆).

26.4.3 Sending a Command to a Secondary

The running primary manipulates the secondary's RXRDY flag and RX buffer in the following manner to send a command to a console secondary.

Programming Note:

The RXRDY flag is a software lock variable; the primary and the secondary must use LDQ_L/STQ_C instructions to set and clear bit n . They must also use MB instructions to order accesses to bin n and commands in the RX buffer. See Chapter 5.

In the following sequence, the console secondary is assumed to have CPU ID = n .

1. The primary examines bit n of the RXRDY bitmask. If the bit is clear, proceed to step 3.
2. The primary polls bit n of the RXRDY bitmask until clear or until some timeout is reached. If a timeout occurs, system software reports an error and takes appropriate action.
3. The primary moves the text of the desired console command into the RX buffer in the secondary's HWRPB slot (the n th per-CPU slot).
4. The primary sets the length of the command into the RXLEN field in the secondary's HWRPB slot (the n th per-CPU slot).
5. The primary issues an MB instruction.
6. The primary sets bit n of the RXRDY bitmask to indicate there is a command waiting.
7. The secondary is assumed to be polling bit n of the RXRDY bitmask.
8. When the secondary notices that bit n of the RXRDY bitmask is set, it issues an MB instruction.
9. The secondary removes the command from its RX buffer.
10. The secondary clears bit n of the RXRDY bitmask, indicating that its RX buffer is again available.
11. The secondary attempts to process the command.

26.4.4 Sending a Message to the Primary

The console secondary (or primary temporarily in console mode) manipulates its TXRDY flag and TX buffer in the following manner to return a message to the running primary.

Programming Note:

The TXRDY flag is a software lock variable; the primary and the secondary must use LDQ_L/STQ_C instructions to set and clear bit n . They must also use MB instructions to order accesses to bit n and messages in the TX buffer. See Chapter 5.

Again, the console secondary is assumed to have CPU ID = n .

1. The secondary examines bit n of the TXRDY bitmask. If the bit is clear, proceed to step 3.
2. The secondary polls this bit until it clears or until a long timeout occurs. (See step 11.)
3. The secondary moves the text of its response message into the TX buffer in the secondary's HWRPB slot (the n th per-CPU slot).
4. The secondary sets the length of the message into the TXLEN field in the secondary's HWRPB slot (the n th per-CPU slot).
5. The secondary issues an MB instruction.

6. The secondary sets bit n of the TXRDY bitmask to indicate there is a message waiting.
7. The secondary issues an MB instruction.
8. If extended RX/TX support is not implemented (see Table 26–2), go to step 10.
If extended RX/TX support is implemented, the secondary sets the TXRDY Summary bit in HWRPB+304.
9. The secondary issues an MB instruction.
10. The console secondary (or the primary temporarily in console mode) issues an interprocessor interrupt to the primary. This is always done; the primary need not poll for bits in the TXRDY bitmask.
11. The secondary polls the TXRDY bitmask until bit n clears or until a long timeout expires. This prevents the secondary from performing any action that might cause the message to be lost before the primary can process it.

Programming Note:

The secondary may be restarted once it has transmitted the error halt message to the primary. However, it must wait for the primary to have a reasonable chance to respond to the interprocessor interrupt and process the message before the restart proceeds, because that message is important visible evidence of the error halt condition. On the other hand, the secondary should not wait too long for the primary to respond because the primary may be affected by the same condition that caused the secondary to error halt. Hence, the need for a timeout that is of reasonable length.

12. As a result of the interprocessor interrupt, the interrupt service routine running on the primary issues an MB instruction.
13. The running primary loads the quadword at HWRPB+304. Whether or not extended RT/TX support is implemented, the primary loads a non-zero value from this location, indicating that a message has been posted for processing by the primary.
14. If extended RX/TX support is not implemented, go to step 16.
If extended RX/TX support is implemented, the primary clears the TXRDY Summary bit in HWRPB+304.
15. The primary issues an MB instruction.
16. The primary notices that bit n of the TXRDY bitmask is set.
17. The primary removes the message from the TX buffer in the n th per-CPU slot.
18. The primary clears bit n of the TXRDY bitmask, indicating that the TX buffer is again available.
19. The primary attempts to process the message.

System Bootstrapping (III)

This chapter describes the net effects of the action of the console to control the system platform hardware. The major system state transitions and the role of the console in controlling those transitions are described in Section 27.1.1. When power is applied to an Alpha system, the console initializes the system as explained in Section 27.2. The console actions necessary to bootstrap system software include processor initialization (Section 27.4.1.5), memory sizing and testing (Section 27.4.1.1), building an initial virtual address space (Section 27.4.1.1), and loading the bootstrap (Section 27.6). The console actions to restart system software are described in Section 27.5.

27.1 Processor States and Modes

27.1.1 States and State Transitions

An Alpha processor can be in one of five major states:

1. Powered off — no system power supplied to the processor
2. Halted — operating system software execution suspended
3. Bootstrapping — attempting to load and start the operating system software
4. Restarting — attempting to restart the operating system software
5. Running — operating system software functioning

As shown in Figure 27-1, the transitions between the major states are determined by the current state and by a number of variables and events, including:

- Whether power is available to the system
- The console `AUTO_ACTION` environment variable, which specifies a "Halt action" (see `CALL_PAL HALT`, Section 6.7.2)
- The console lock setting
- The Bootstrap-in-Progress (BIP) flags
- The Restart-Capable (RC) flags
- Processor error halts
- The `CALL_PAL HALT` instruction
- Console commands

Figure 27–1: Major State Transitions

Action Causing Transition to Final State	Initial State				
	Off	Halted	Booting	Restart	Running
Powerfail	Off	Off	Off	Off	Off
A and Power Restored	Halted				
B and Power Restored	Booting				
C and Power Restored	Restart				
BOOT and Console Is Locked		Booting			
START or CONTINUE (and) Console Is Unlocked		Running			
Bootstrap Fails or D			Halted		
Bootstrap Succeeds			Running		
D				Halted	
Restart Fails				Booting	
Restart Succeeds				Running	
A and Processor Halts or D					Halted
B and Processor Halts					Booting
C and Processor Halts					Restart

Final State

Key to Figure 27–1

- A Console is unlocked and AUTO_ACTION is "HALT".
- B Console is unlocked and AUTO_ACTION is "BOOT".
- C Console is unlocked and AUTO_ACTION is "RESTART" or console is locked.
- D Console is unlocked, the processor is forced into console I/O mode.

To effect major state transitions, the console obeys these rules:

- If the console is unlocked when power is restored or when the processor halts, enter the state selected by the console AUTO_ACTION environment variable.
- If the console is locked when power is restored or when the processor halts, attempt a processor restart.
- When processor restart fails, attempt a bootstrap of that processor. One cause of a failed restart is the processor's RC flag being clear when the console attempts the restart.
- When system bootstrap fails, halt. One cause of a failed bootstrap is the processor's BIP flag being set before the console attempting the bootstrap. Only the processor that failed bootstrap will halt.
- When system bootstrap or processor restart succeeds, the processor starts running.
- When the primary processor is halted and the console is unlocked, the console BOOT command causes a system bootstrap.
- When a secondary processor is halted and the console is unlocked, the console START –CPU command causes the console to attempt to start that processor running.
- When a processor is halted and the console is unlocked, the console CONTINUE command causes the processor to continue running as though no halt was incurred.

Note:

Continuation of system software by the CONTINUE command causes the processor to continue running as though no HALT has occurred.

- If the console is unlocked and a specified processor is running or booting or restarting, that processor is halted by a console HALT –CPU command.

Implementation Note:

In an embedded console implementation, the primary processor must be forced into the console I/O mode before issuing the HALT –CPU command.

27.1.2 Major Modes

In addition to the major states, the console and processor are described as being in one of three modes:

1. Program I/O mode

The processor is running. The processor interprets instructions, services interrupts and exceptions, and initiates I/O operations under the control of the operating system.

2. Console I/O mode

The processor is halted or bootstrapping or restarting. The console provides control over the system; the operating system has either relinquished control or has yet to gain control. The operating system does not service interrupts or exceptions or initiate I/O operations. The actions of the console are determined by internal console state and commands from the console operator.

3. Console Initialization mode

The console has yet to acquire control of the processor. The console itself may also require initialization, such as when power is first applied to the system.

A given processor may be in one of four modes:

- Primary processor in program I/O mode or "running primary"
- Primary processor in console I/O mode or "console primary"
- Secondary processor in program I/O mode or "running secondary"
- Secondary processor in console I/O mode or "console secondary"

As noted in Section 25.1, implementations must include a mechanism to force a processor executing in program I/O mode into console I/O mode.

27.2 System Initialization

An Alpha system must be initialized when power is restored. System initialization also occurs as the result of a system bootstrap when the BOOT_RESET environment variable is set to "ON", or as the result of the console INITIALIZE command. Initialization involves all implementation-specific, system-wide actions necessary to allow the system to boot system software on the primary processor. Table 27–1 summarizes the effects of initialization as seen by system software.

Initialization may include initialization of the console itself. During console initialization, the console must build the HWRPB and all associated data structures necessary to permit the console to accept console commands and boot system software.

System initialization may also include any necessary system bus, processor, or I/O device initialization. The initialization of a processor performed as part of system initialization is not necessarily that performed just before transfer of control to the operating system bootstrap. See Section 27.4.1.5 for a description of processor initialization as seen by system software.

Table 27–1 Effects of Power-Up Initialization

Processor State	Initialized State
BIP and RC flags	Cleared
Reason for halt code	'0' (bootstrap)
Integer and floating-point registers	UNPREDICTABLE
System memory	Unaffected if preserved by battery backup; otherwise, UNPREDICTABLE
Environment variables	Unaffected if nonvolatile; otherwise, set to default
BB_WATCH	Unaffected
I/O device registers	UNPREDICTABLE

27.3 PALcode Loading and Switching

27.3.1 PALcode Loading

The console loads PALcode into good memory within a memory cluster that is not available to system software. If PALcode scratch space is required, the console allocates good memory within a memory cluster that is not available to system software. PALcode memory and scratch space are at least page aligned. The console records the starting physical address and length of PALcode memory and scratch space and then sets the PALcode Memory Valid (PMV) flag in the per-CPU slot of the primary processor. The PMV flag indicates that the PALcode descriptors are valid.

After PALcode loading and initialization, the console sets the PALcode Loaded (PL) and PALcode Valid (PV) flags in the primary's per-CPU slot. The PL flag indicates that PALcode has been loaded; the PV flag indicates that any necessary PALcode initialization has been performed.

PALcode loading and initialization are implementation specific. The PALcode source may be a special console device, ROM, a system device, a communications line, or any other implementation-specific source. The state of the console and system must be such that the source is accessible. The console determines the PALcode variant in an implementation-specific fashion; console implementations that are dependent on a given variant load that variant. Console and platform implementations may select any PALcode variant and may load multiple PALcode variants.

Note:

Tru64 UNIX and Alpha Linux support PALcode switching but do not support PALcode loading. Any platform that supports either operating system must either use the Tru64 UNIX or Alpha Linux variant as the default or must load (but need not switch to) that variant before system bootstrap.

The means by which any PALcode internal state is initialized is implementation specific.

27.3.2 PALcode Switching

PALcode switching is accomplished when one ("current") PALcode transfers control to another ("new") PALcode. PALcode switching can be initiated by the console or the operating system software.

Note:

OpenVMS does not support PALcode switching. Any platform that supports OpenVMS must either use the OpenVMS variant as the default or must switch to the OpenVMS variant before system bootstrap.

PALcode switching is performed by PALcode without intervention from the console or operating system software. The current PALcode must be able to locate the new PALcode image. The new PALcode may perform minimal sanity checks.

To support PALcode switching, all PALcode images must implement a PALcode switching entry point at the image base (offset 0). During PALcode switching, the new PALcode image receives control from the current PALcode image at this offset.

For the purposes of switching, a PALcode image is identified by one of the following:

- PALcode variant

PALcode variants are in the range $0 < \text{variant} < 256$ and permit switching between cooperating, previously loaded PALcode images. PALcode variants are interpreted by the current PALcode without assistance from the console or operating system.

- The physical address of the switching entry point.

Entry point addresses are used whenever the operating system or console must load a PALcode image. Entry point addresses must meet the alignment requirements of the processor implementation and may occupy the lowest memory page.

System software initiates PALcode switching during system bootstrap whenever the variant required is not identical to that supplied by the console. Once a new variant has been established by system software, the console must restore that variant across all subsequent transitions from console I/O mode to program I/O mode. The console must ensure that the system software PALcode variant appears unchanged when:

1. A processor is restarted.
2. A secondary processor is started.
3. The operator forces a processor into console I/O mode, then continues program execution (HALT followed by CONTINUE).
4. System software invokes a callback routine that requires transition to console I/O mode.

System software is never required to restore a PALcode variant. The console may switch PALcode at entries to console I/O mode, but must restore the variant established by system software at subsequent re-entry to program I/O mode.

27.3.2.1 PALcode Switching Procedure

PALcode switching proceeds as follows:

1. The current PALcode is entered by the CALL_PAL SWPPAL instruction. The PALcode image identifier (variant or switching entry point address) is contained in R16. Registers R17 through R21 contain parameters that are passed without change to the new PALcode image. The interpretation of R17 through R21 is specific to the new PALcode image.
2. If the current PALcode is not supplied by Compaq and does not support PALcode switching, the current PALcode sets R0 = 1 and returns from the CALL_PAL SWPPAL instruction.
3. The current PALcode determines if R16 contains a PALcode variant or switching entry point address. If the latter, execution continues at step 7.
4. The current PALcode validates the PALcode variant. If unsuccessful, the operation fails, the current PALcode sets R0 = 1 and returns from the CALL_PAL SWPPAL instruction.
5. The current PALcode determines if the PALcode associated with the PALcode token has been loaded. If not, the operation fails, the current PALcode sets R0 = 2 and returns from the CALL_PAL SWPPAL instruction.
6. The current PALcode determines the base physical address associated with the PALcode token.
7. The current PALcode branches to the new PALcode image at the switching entry point (physical) address determined in step 3 or 6.
8. The new PALcode performs any necessary implementation-specific PALcode initialization.
9. The new PALcode invalidates all TB entries and establishes the new memory management algorithm. (For example, PALcode for Tru64 UNIX and Alpha Linux loads the VPTB with a value supplied to the CALL_PAL SWPPAL instruction.)
10. The new PALcode performs any implementation-specific actions using the entry parameters contained in R17 through R21. The resulting changes in processor state are summarized for each PALcode variant in Section 27.3.2.3.
11. The new PALcode clears R0 and passes control to the code thread determined by the entry parameters. Control is always passed in kernel mode with interrupts disabled or blocked.

If a hardware failure occurs when accessing any of the addresses specified by the calling arguments or other dependent locations, a hardware reset and system initialization are performed.

Implementation Note:

A common implementation is that the switching entry point is identical to the hardware reset entry. PALcode must distinguish the two cases. In the case of hardware reset, PALcode must perform any necessary hardware initialization and pass control to the console. In the case of switching, PALcode must pass control to the code thread determined by the entry parameters.

Notes:

- System software must update the PALcode revision field (SLOT[168]) after PALcode switching. The console uses that field to determine if PALcode must be switched (to the system software-specific image) before passing control on system restarts.

Similarly, system software may need to update the PALcode revision field in the per-CPU slot (SLOT[168]) of each secondary processor before starting the secondary. There is only one system restart routine. The console uses the PALcode revision field to determine if PALcode must be switched (to the system software-specific image) before passing control on secondary processor starts.

- PALcode switching is initiated by invoking the CALL_PAL SWPPAL instruction. Before invoking SWPPAL, the caller should ensure that the system is quiescent. It is recommended that SWPPAL be invoked with interrupts either disabled or blocked. After a successful PALcode switch, the operating system may need to update the VPTB field in the HWRPB or restart HWPCB in each per-CPU slot.
- PALcode switching does not implicitly load PALcode. During system bootstrap, the operating system must ensure that the desired PALcode variant is loaded. If loading is required, the operating system must allocate sufficient physically contiguous physical memory for the new PALcode image and any additional PALcode scratch space, then load the PALcode image in an implementation-specific manner.
- After a PALcode switch, the operating system may need to invoke the FIXUP console callback routine. FIXUP must be invoked after any operation that affects virtual address translation and before subsequent invocations of other callback routines. See Section 26.3.7.2.

27.3.2.2 Specific PALcode Switching Implementation Information

OpenVMS does not currently support PALcode switching. Tru64 UNIX and Alpha Linux supports PALcode switching as shown in Table 27–2.

Table 27–2: Tru64 UNIX and Alpha Linux PALcode Switching

Register	CALL_PAL swppal Parameter Usage
R17 (a1)	New PC
R18 (a2)	New PCBB
R19 (a3)	New VPTB

27.3.2.3 Processor State at Exit from PALcode Switching Instruction

Table 27–3: Processor State at Exit from swppal

Processor State		At Exit from swppal:
ASN	Address space number	ASN in PCB passed to swppal
FEN	Floating enable	FEN in PCB passed to swppal
Integer and floating-point registers		UNPREDICTABLE, except SP and R0
IPL	Interrupt priority level	7
KSP	Kernel stack pointer	KSP in PCB passed to swppal
MCES	Machine check error summary	Zero
Other IPRs		UNPREDICTABLE
PC	Program counter	PC passed to swppal
PCBB	Privileged context block	Address of PCB passed to swppal
PS	Processor status	IPL=7, CM=K
PTBR	Page table base register	PTBR in PCB passed to swppal
R0		Zero
Sysvalue	System value	Unchanged
Unique	Processor unique value	Unique in PCB passed to swppal
VIRBND	Virtual Boundary Register	–1
WHAMI	Who-Am-I	Unchanged

27.4 System Bootstrapping

This section describes the operations performed by the Alpha console to locate, load, and transfer control to a primary bootstrap. The responsibilities of the console and the initial state seen by system software are presented for multiprocessor and uniprocessor environments. The actions of the console for cold bootstrap (full hardware initialization) and warm bootstrap (partial hardware initialization) are described.

A system bootstrap can occur as the result of a powerfail recovery, a processor halt, or an INITIALIZE or BOOT console command. See Section 27.1.1 for a complete description of these state transitions.

27.4.1 Cold Bootstrapping in a Uniprocessor Environment

This section describes a cold bootstrap in a uniprocessor environment. A system bootstrap is a cold bootstrap when any of the following occur:

- Power is first applied to the system.
- The bootstrap is requested by system software.

- A console INITIALIZE command is issued and the AUTO_ACTION environment variable is set to "BOOT".
- The BOOT_RESET environment variable is set to "ON".

The console must perform the following steps in the cold bootstrap sequence.

1. Perform a system initialization
2. Size memory
3. Test sufficient memory for bootstrapping
4. Load PALcode
5. Build a valid Hardware Restart Parameter Block (HWRPB)
6. Build a valid set of Memory Cluster Descriptors
7. Initialize bootstrap page tables and map initial regions
8. Locate and load the system software primary bootstrap image
9. Initialize processor state on all processors
10. Transfer control to the system software primary bootstrap image

The steps leading up to the transfer of control to system software may be performed in any order. The final state seen by system software is defined, but the implementation-specific sequence of these steps is not. Before beginning a bootstrap, the console must clear any internally pended restarts to any processor.

27.4.1.1 Memory Sizing and Testing

Memory sizing is the responsibility of the console. The console must also test sufficient memory to permit control to be passed to the primary bootstrap image. The results of console memory sizing and testing are passed to system software using memory cluster descriptors.

Each memory cluster descriptor describes a physically contiguous extent of physical memory that contains no holes. The memory within a cluster is either available to system software or reserved for console use. Usage within a cluster cannot be mixed; if the cluster contains a page reserved for console use, system software cannot allocate any page within the cluster. The memory cluster descriptor contains a cluster usage field that indicates the cluster availability to system software. The primary bootstrap image must reside in clusters available to system software.

The memory within each cluster may be fully tested, partially tested, or untested by the console. If the memory is untested, no cluster memory bitmap is built. The console must test enough memory to allow the primary bootstrap image to be loaded and control to be passed to that image. This memory includes:

- PALcode memory and scratch areas
- CPU logout areas
- Memory bitmaps
- HWRPB and all offset blocks
- Console CRB map entries
- Bootstrap address space page tables

- Primary bootstrap image
- One page for the initial bootstrap stack

Any additional memory testing by the console is implementation specific. It is the responsibility of system software to test any memory not tested by the console.

A cluster bitmap is built if the cluster is available to system software and the console tests any memory within the cluster. Each page in the cluster is represented by a bit in the bitmask. A '1' in the bitmap means that the corresponding page is "good"; the page was tested without error. A '0' in the bitmap means that the corresponding page is "bad"; the page is either untested or was tested but encountered correctable (Corrected Read Data) errors or hard (Read Data Substitute) errors.

Cluster bitmaps must be at least quadword aligned and must be an integral number of quadwords; any unused bits in the highest addressed quadword must be zero.

Implementation Notes:

Every implementation cannot be required to test all of memory before booting the operating system. Partial memory testing is recommended whenever testing is time-consuming and would significantly delay the bootstrapping process; the choice is implementation specific. The high-water mark mechanism allows implementations to completely size memory without testing all of it and indicate to the operating system where testing ended.

Clusters reserved for the use of the console and PALcode do not have associated bitmaps.

The console does not alter the Memory Cluster Descriptors or any bitmaps across warm bootstraps. This permits system software to propagate information on system software memory testing and intermittent errors across operating system bootstraps. For example, system software could set the "bad" bit of a page that incurred repeated CRD errors.

27.4.1.2 Passing Memory Cluster Descriptors to System Software

Memory cluster descriptors are passed to system software in one of two ways:

- They may be statically built into the Memory Data Descriptor (MEMDSC) table located by HWRPB[200]. This is used by all platforms supporting HWRPB Revision 11 or earlier. See Section 27.4.1.2.1.
- Starting at HWRPB Revision 12, they may be distributed (dynamically built and deleted) by using a combination of the MEMDSC table located by HWRPB[200] and the FRU table located by HWRPB[216]. See Sections 26.1.5 and 27.4.1.2.2.

The format of a static memory cluster descriptor in the MEMDSC table differs from that of a distributed memory cluster descriptor in the FRU table.

27.4.1.2.1 Static Memory Clusters in the MEMDSC Table

The memory data descriptor (MEMDSC) table contains one or more static memory cluster descriptors. Static cluster descriptors are ordered by increasing physical address; the range of PFNs described by cluster n is of lower address than the range of PFNs described by cluster $n+1$.

The MEMDSC table must be quadword aligned and both physically and virtually contiguous. The MEMDSC table format is shown in Figure 27–2; the memory cluster descriptor format is shown in Figure 27–3. The size of the MEMDSC table can be determined by the number of clusters contained in MEMDSC[16]. The size of the table and the offset to the last quadword of the table are given by:

$$\text{MEMDSC_SIZE} = ((7 * \text{MEMDSC}[10_{16}]) + 3) * 8$$

$$\text{MEMDSC_END} = \text{MEMDSC_SIZE} - 8$$

Figure 27–2 Memory Data Descriptor (MEMDSC) Table

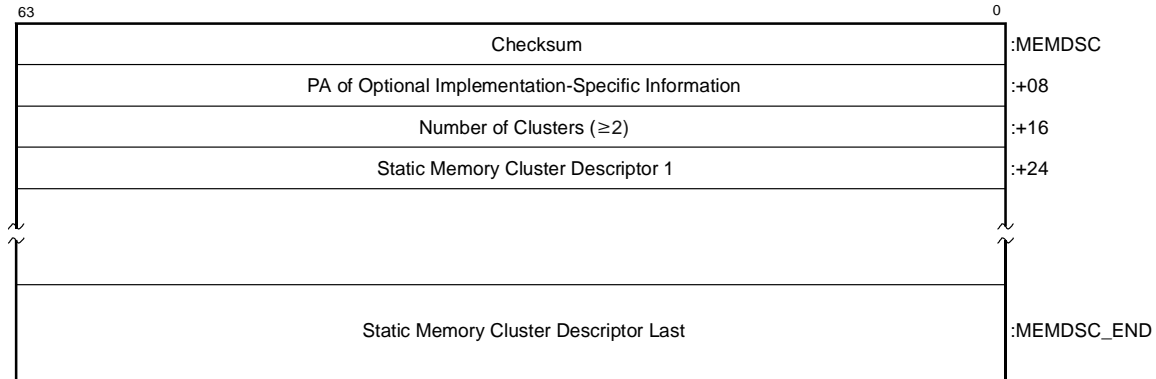


Figure 27–3 Static Memory Cluster Descriptor

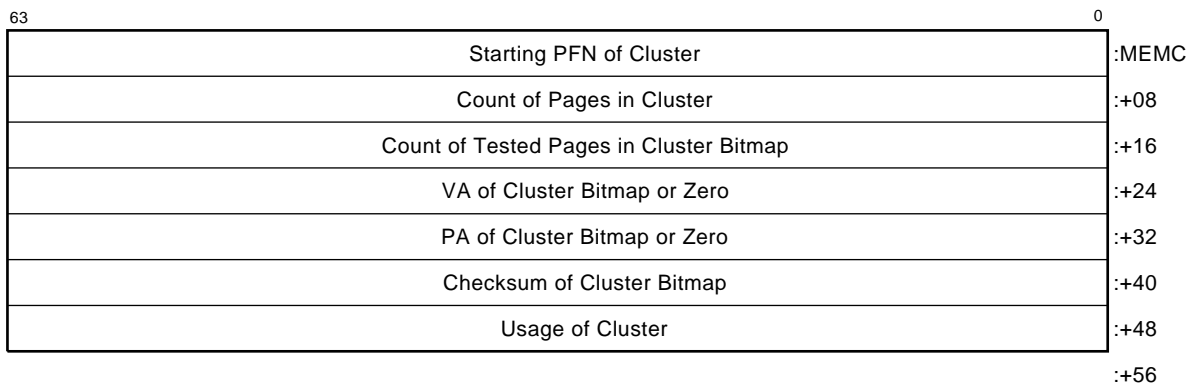


Table 27–4 Memory Data Descriptor Table Fields

Offset	Description
MEMDSC	CHECKSUM — Checksum of all the quadwords from offset MEMDSC+8 through MEMDSC_END. Computed as a 64-bit sum, ignoring overflows. The checksum does not include any of the cluster bitmaps or any optional implementation-specific data.

Table 27–4 Memory Data Descriptor Table Fields (Continued)

Offset	Description
+08	IMP_DATA_PA — Physical address of additional implementation-specific information (if any). If no additional implementation-specific information exists, the field must contain a zero.
+16	CLUSTERS — Number of clusters in the memory data cluster descriptor table. Unsigned integer greater than or equal to two (at least one cluster for console memory and once cluster for software memory, with no null descriptor present). See Figure 27–4.
+24	CLUSTER — Each static memory cluster descriptor describes an extent of physical memory. See Figure 27–3.

Table 27–5 Static Memory Cluster Descriptor Fields

Offset	Description
MEMC	PFN — Starting PFN of the memory cluster.
+08	PAGES — Number of pages in the memory cluster. Unsigned integer.
+16	TESTED_PAGES — Number of tested memory pages in the cluster. If only a limited extent of the cluster memory was tested, a bitmap is built, and this high-water mark indicates the number of pages that were tested. The tested range is always the lowest-ordered part of the range within this cluster.
+24	BITMAP_VA — Starting virtual address of the cluster memory testing bitmap in the bootstrap address space. If the memory is untested, no bitmap is built and this field is set to zero.
+32	BITMAP_PA — Starting physical address of the cluster memory testing bitmap. If the memory is untested, no bitmap is built and this field is set to zero.
+40	BITMAP_CHECKSUM — Checksum of the cluster memory testing bitmap. Computed as a 64-bit sum, ignoring overflows, over the TESTED_PAGES active bits only.
+48	USAGE — Indicates whether the cluster is available for use by system software. <ul style="list-style-type: none"> • If USAGE<0> is ‘0’, system software may allocate and use the cluster. • If USAGE<0> is ‘0’ and USAGE<1> is ‘1’, the cluster is available for use by the system software, but is in nonvolatile memory. • If USAGE<0> is ‘1’, the cluster is reserved for console use and must not be allocated by system software. • USAGE<63:2> should be zero.

27.4.1.2.2 Distributed Memory Cluster Descriptors in the FRU Table

HWRPB Revision 12 introduces an option for presenting the results of memory sizing and testing to system software. This option distributes the memory cluster descriptors to the FRU table (Section 26.1.5) where they may be organized more flexibly than is possible with only the MEMDSC table. Such flexibility better supports memory reconfiguration while system software (one or more "Instances" of which) is active, including:

- Partitioning memory such that multiple Instances may execute simultaneously, each within a designated "Instance-private" complement (partition) of memory
- Dynamic reallocation of memory among partitions while Instances are active
- Designating memory as shareable among a cooperative community of Instances
- Hot-addition of memory to a running Instance
- Hot-removal of memory from a running Instance

This presentation option leverages the MEMDSC table. Instead of containing static memory cluster descriptors, the MEMDSC table includes only a 'null' memory descriptor that effectively points to memory cluster descriptors that are distributed to the FRU table. Those distributed memory cluster descriptors are linked into a list that altogether describes the physical memory available to an Instance. The console may locate the distributed memory cluster descriptors anywhere within the FRU table; they are not packed together nor arranged in any particular order as is the case for the static descriptors embedded in the MEMDSC table.

Figure 27–4 shows and Table 27–6 describes the MEMDSC table format when used to indicate that memory cluster descriptors are distributed to the FRU table.

Figure 27–4 MEMDSC Table with Null Memory Cluster Descriptor

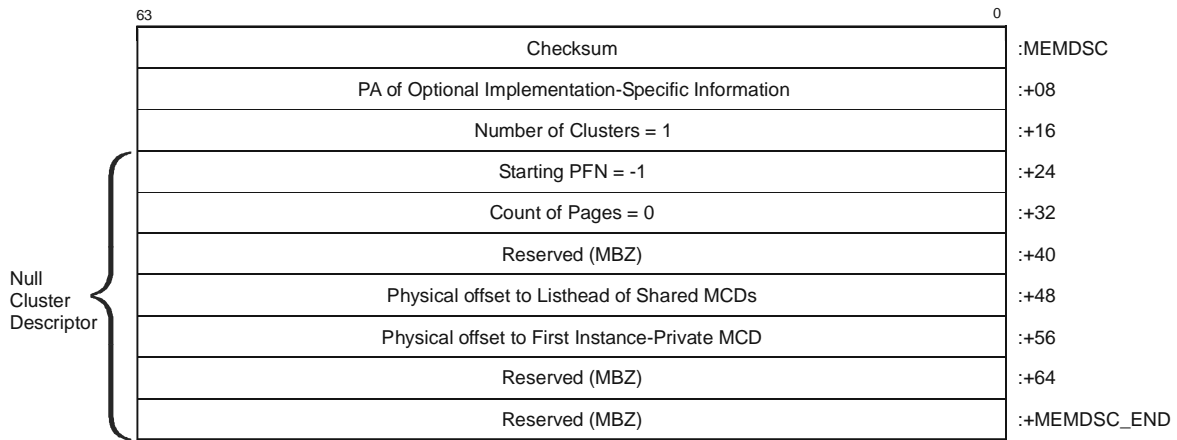


Table 27–6 MEMDSC Table Fields with Null Memory Cluster Descriptor

Offset	Description
MEMDSC	CHECKSUM ^{1,2} – Checksum of all the quadwords from offset MEMDSC+8 through MEMDSC_END. Computed as a 64-bit sum, ignoring overflows. The checksum does not include any optional implementation-specific data.
+08	IMP_DATA_PA ^{3,2} – Physical address of additional implementation-specific information (if any). If no additional implementation-specific information exists, the field must contain a zero.
+16	CLUSTERS ^{3,4} – Number of clusters in the Memory Data Descriptor table. Unsigned integer equal to one (a single null cluster descriptor present; i.e. no static memory cluster descriptors (Figure 3-3) will be present).
+24	PFN ^{3,4} – Starting PFN of the memory cluster set to -1, an invalid PFN.

Table 27–6 MEMDSC Table Fields with Null Memory Cluster Descriptor

+32	PAGES ^{3,4} – Number of pages in the memory cluster set to zero.
+40	Reserved for future use - MBZ
+48	SHARED_MCDS ³ – Physical offset to the listhead for distributed memory cluster descriptors that describe memory that has been designated as shareable among a cooperative community of Instances. Offset is from the base of the FRU table as located by HWRPB+216. Signed integer.
+56	PRIVATE_MCDS ^{3,1} – Physical offset to the first distributed memory cluster descriptor that describes memory reserved for use by this instance of system software. Offset is from the base of the FRU table as located by HWRPB+216. Signed integer.
+64	Reserved for future use - MBZ
MEMDSC_END	Reserved for future use - MBZ

¹ May be modified by the console.

² May be modified by system software.

³ Initialized by the console at cold system bootstrap only. Preserved unchanged by the console at all warm system bootstraps.

⁴ May be used to discern the presence of a null memory cluster descriptor.

Figure 27–5 shows and Table 27–7 describes the format of a distributed memory cluster descriptor. This differs from the format of the static memory cluster descriptor depicted in Figure 27–3.

Figure 27–5 Distributed Memory Cluster Descriptor

Checksum		:DMEMC
Physical offset to next MCD in List or -1		:+08
Count of Pages in Cluster	Starting PFN of Cluster	:+16
Usage of Cluster	Count of Tested Pages in Cluster Bitmap	:+24
PA of Cluster Bitmap or Zero		:+32
Checksum of Cluster Bitmap		:+40
Reserved (MBZ)		:+48
Reserved (MBZ)		:+56
Reserved (MBZ)		:+64
Reserved (MBZ)		:+DMEMC_END

Table 27–7 Distributed Memory Cluster Descriptor Fields

Offset	Description
DMEMC	CHECKSUM ^{1,2} – Checksum of all the quadwords from offset DMEMC+8 through DMEMC_END. Computed as a 64-bit sum, ignoring overflows. The checksum does not include any of the cluster bitmaps.
+08	OFFSET ^{3,1} – Physical offset to the next distributed memory cluster descriptor in the list, or –1 if none. Offset is from the base of the FRU table as located by HWRPB+216. Signed integer.
+16	PFN ^{3,1} – Starting PFN of the memory cluster.
+20	PAGES ^{3,1} – Number of pages in the memory cluster. Unsigned integer.
+24	TESTED_PAGES ^{3,2} – Number of tested memory pages in the cluster. If only a limited extent of the cluster memory was tested, a bitmap is built, and this high-water mark indicates the number of pages that were tested. The tested range is always the lowest-ordered part of the range within this cluster.

Table 27–7 Distributed Memory Cluster Descriptor Fields (Continued)

+28	USAGE ^{3,1} – Indicates characteristics of this memory cluster. Bits Meaning 31:4 Reserved for future use and should be zero. 3 When set, the cluster descriptor is FIXED such that it cannot be deleted or agglomerated with another cluster descriptor while this instance of system software remains active. The console may set FIXED for a cluster reserved for console use to indicate to software that the console is incapable of evicting the contents of that cluster. 2 Indicates the list that includes this cluster descriptor. When clear, indicates the PRIVATE_MCDS list (see Figure 27–4); when set, indicates the SHARED_MCDS list. 1 When set, the cluster is in nonvolatile memory. 0 When clear, system software may allocate and use the cluster. When set, the cluster is reserved for console use and must not be allocated by system software.
+32	BITMAP_PA ³ – Starting physical address of the cluster memory testing bitmap. If the memory is untested, no bitmap is built and this field is set to zero.
+40	BITMAP_CHECKSUM ^{3,2} – Checksum of the cluster memory testing bitmap. Computed as a 64-bit sum, ignoring overflows, over the TESTED_PAGES active bits only.
+48 through DMEMC_END	Reserved for future use - MBZ

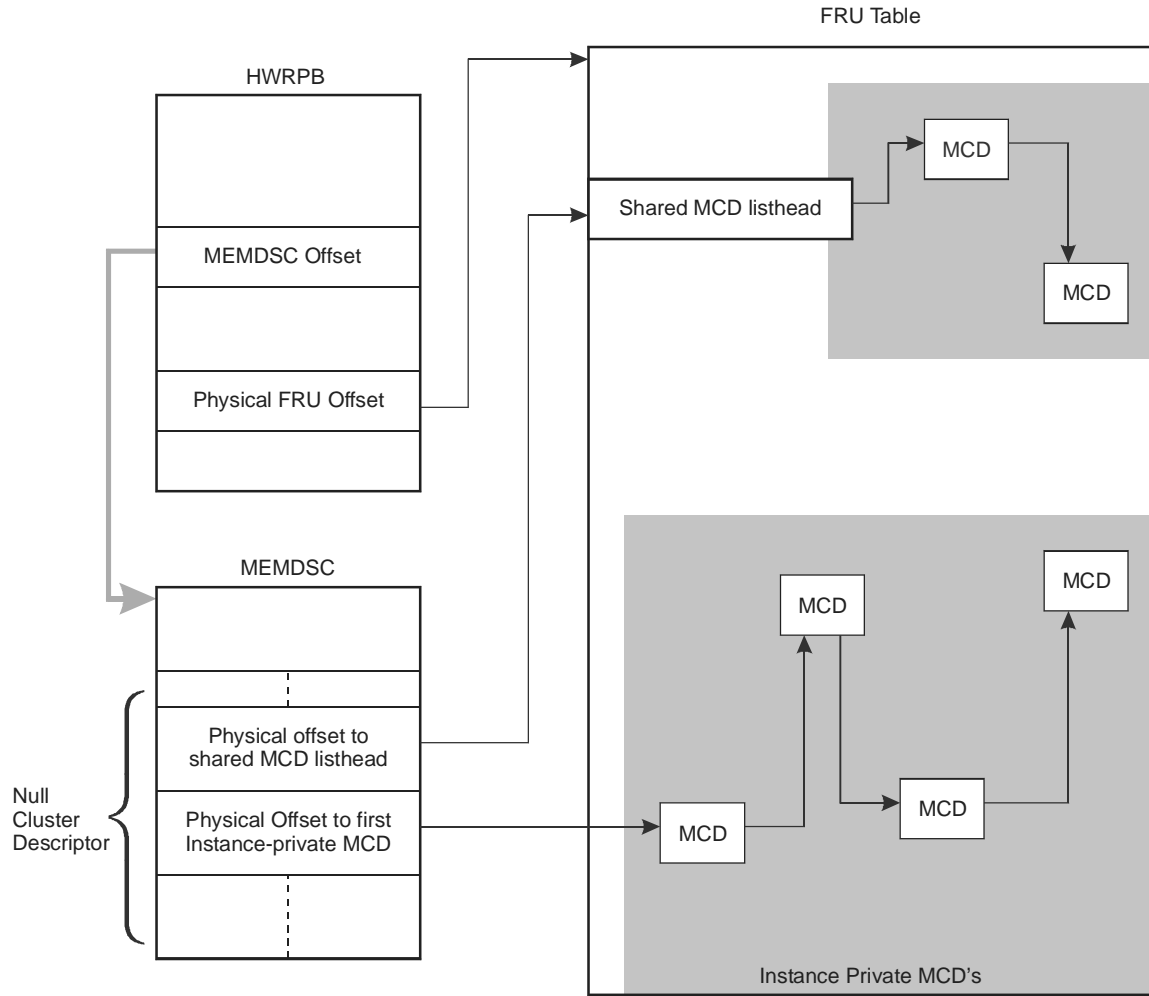
¹ May be modified by the console.

² May be modified by system software.

³ Initialized by the console at cold system bootstrap only. Preserved unchanged by the console at all warm system bootstraps.

Figure 27–6 illustrates how these structures are linked together.

Figure 27–6 Distributed Memory Cluster Descriptors



27.4.1.3 Bootstrap Address Space

All system software, including the primary bootstrap image, runs in a virtual memory environment. The console creates the initial page tables that define the initial bootstrap address space for the primary bootstrap. System software may replace this bootstrap address space at any time after the console passes control to the primary bootstrap image.

The bootstrap address space consists of four regions. All regions must be located in good memory within clusters that are available to system software. The regions are:

Region 0

This region maps console or PALcode data structures that must be shared with system software. These structures include the HWRPB in its entirety, all physically contiguous blocks located by HWRPB offsets, the console callback routines, and all tested memory bitmaps located by static memory descriptors in the Memory Data Descriptor (MEMDSC) table. Region 0 begins at address 256MB, virtual address 0000 0000 1000 0000₁₆. The starting address of the HWRPB is the base of Region 0.

Region 1

The primary bootstrap image is loaded into this region. The region must be at least large enough to load system software plus three pages. The three additional pages are used as an initial bootstrap stack and stack guard pages. The stack guard pages are virtually adjacent to the bootstrap stack page and marked no-access. All other pages in the region are mapped and valid. Region 1 begins at address 512MB, virtual address 0000 0000 2000 0000₁₆.

Software Note:

This region must be set to the size of the primary bootstrap image plus 3 pages for OpenVMS and at least 256K bytes for Tru64 UNIX and Alpha Linux.

Region 2

This region, or "page table space," contains the bootstrap address space page tables. Region 2 begins at address 1GB, virtual address 0000 0000 4000 0000₁₆. The range depends on the page size:

Page Size	Page Table Space Address Range
8KB	1GB to 1GB+8MB
16KB	1GB to 1GB+16MB
32KB	1GB to 1GB+32MB
64KB	1GB to 1GB+64MB

This region includes the Level 2 and Level 3 page tables used to map all three regions comprising bootstrap address space. The Level 2 page table maps itself as a Level 3 page table. The address of the Level 2 page table page and the PTE within the page that is used for self-mapping also depend on the page size.

Page Size	Virtual Address of Level 2 Page Table	L2PTE Number Used for Self-Mapping
8KB	1GB+1MB	128
16KB	1GB+512KB	32
32KB	1GB+256KB	8
64KB	1GB+128KB	2

Implementation Note:

Region 2 allows the primary bootstrap code to start with 32-bit pointers that execute in a 32-bit context. Thus, Region 2 allows primary bootstrap software to be written with 32-bit-oriented language compilers.

The initial page tables that map the virtual address regions are shown in Figure 27–7 and illustrated in Figure 27–8.

Region 3

This region maps the entire page table structure, including all levels of page table, that would be required to map the entire virtual address space supported by this implementation. The Level 1 page table is self-mapped by the second PTE in the page.

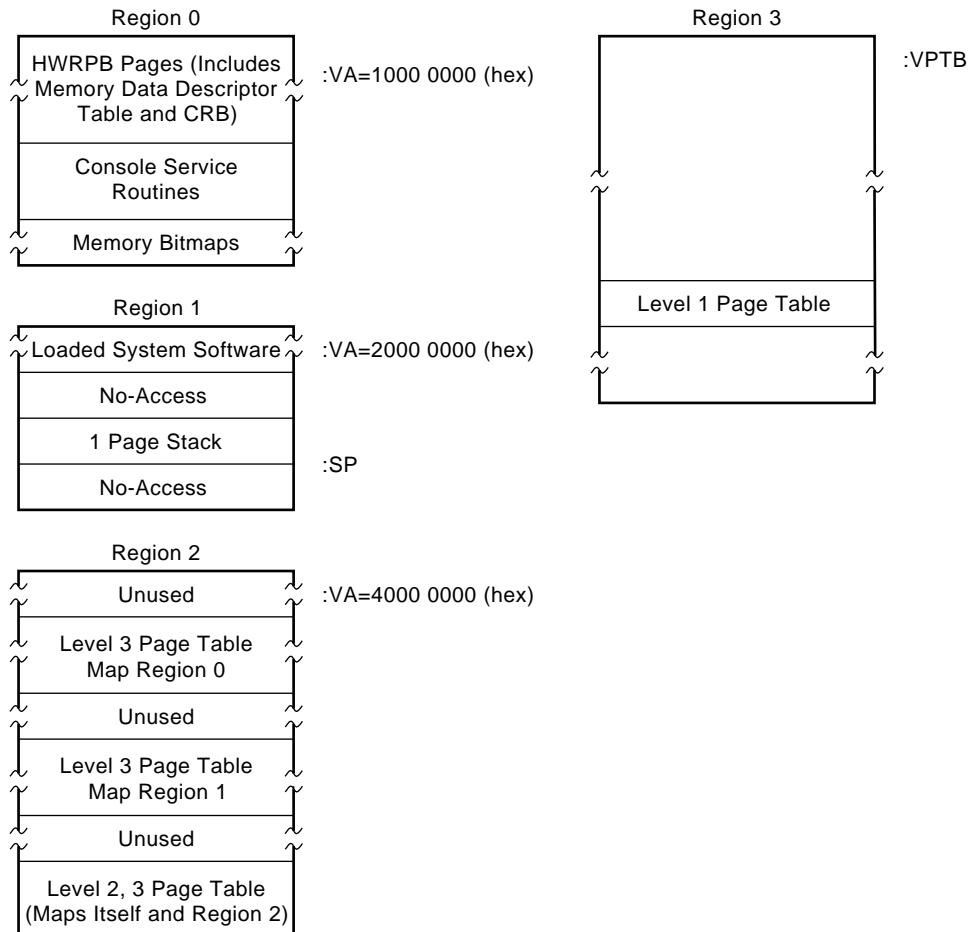
Region 3 exists to support virtual page table lookup for Translation Buffer misses. Region 3 exists at a virtual address that is inaccessible to code that is compiled to support only a 32-bit virtual address space. As such, Region 3 is not the primary page table space that is presented to bootstrap software.

Programming Note:

Due to the self-mapping, Region 3 maps all page table pages. The Level 2 and Level 3 page table pages are in both Region 2 and Region 3.

Page Size	Virtual Address of Page Table Space (VPTB)
8KB	8GB
16KB	64GB
32KB	512GB
64KB	4TB

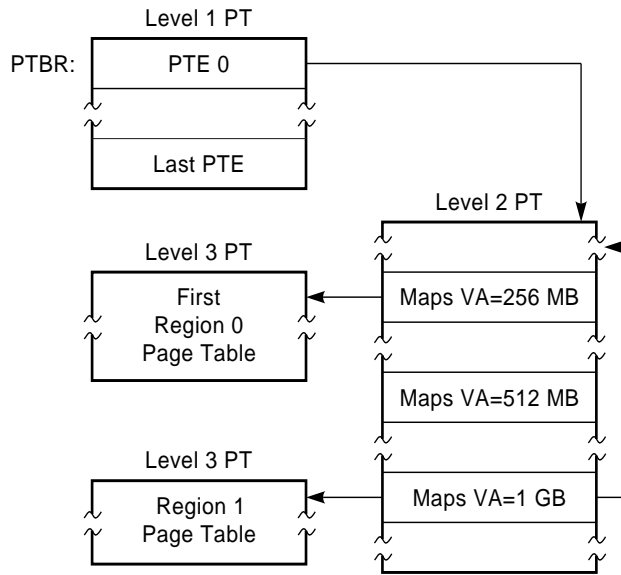
Figure 27–7: Initial Virtual Memory Regions



All valid pages allow read/write access from kernel mode and deny all access from other modes. All fault bits (FOR, FOW, FOE) are clear, as well as Address Space Match (ASM) and Granularity Hint (GH).

The self-mapping of the Level 2 page table excludes the Level 1 page table from Region 2. In this case, the Level 1 page table has two active PTEs. The first L1PTE points to the PFN of the Level 2 page table page, which maps page table space (Region 2). The second L1PTE contains the PFN of the Level 1 page table itself, thus defining Region 3. Only these two entries within the Level 1 page table are valid; all other Level 1 PTEs are zero.

Figure 27–8: Initial Page Tables

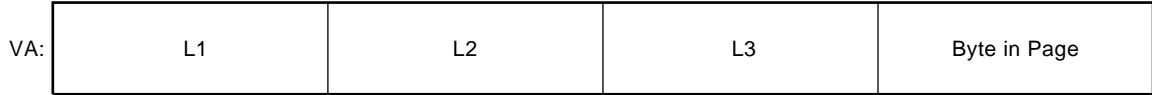


The level 2 PT maps Region 2 (page table space) at 1 GB. The level 2 PT maps itself as its own level 3 PT.

The level 1 PT is not mapped.

The self-mapping of the Level 2 page table also causes the addresses of the Level 2 and Level 3 PTEs for a given virtual address to be functions of that address. For every virtual address within the bootstrap address space, there is exactly one location within page table space for the Level 2 PTE that maps that virtual address, and exactly one location for the Level 3 PTE that maps that virtual address.

Thus, the Level 2 and Level 3 PTE virtual addresses for a given virtual address (VA) within bootstrap address space can be calculated given the page size. The following bit range definitions provide convenient notation for referring to the constituent parts of a virtual address. For example, VA<L2> is equivalent to VA<32:23> for an 8K byte page size.



Page Size	L1	L2	L3
8KB	42:33	32:23	22:13
16KB	46:36	35:25	24:14
32KB	50:39	38:27	26:15
64KB	54:42	41:29	28:16

The base of page table space is a constant value:

1. $PT_Base = 1GB$

The virtual address of the Level 3 PTE (L3PTE_VA) of any virtual address (VA) is given by:

2. $L3PTE_VA(VA) = PT_Base + (page_size * VA\langle L2 \rangle) + (8 * VA\langle L3 \rangle)$

Thus, the virtual address of the Level 3 PTE that maps the lowest address of page table space is given by:

$$L3PTE_VA(PT_Base) = PT_Base + (page_size * PT_Base\langle L2 \rangle)$$

Since the Level 2 page table is self-mapped, the above is also the base virtual address of the Level 2 page table. Thus:

3. $L2PT_Base = PT_Base + (page_size * PT_Base\langle L2 \rangle)$

Finally, the virtual address of the Level 2 PTE (L2PTE_VA) of any virtual address (VA) is given by:

$$L2PTE_VA(VA) = L2PT_Base + (8 * VA\langle L2 \rangle)$$

4. $L2PTE_VA(VA) = PT_Base + (page_size * PT_Base\langle L2 \rangle) + (8 * VA\langle L2 \rangle)$

27.4.1.4 Bootstrap Flags

The Bootstrap-in-Progress (BIP) and Restart-Capable (RC) processor state flags in the primary processor's per-CPU slot are used to detect failed bootstraps. If the primary re-enters console I/O mode while the BIP flag is set and the RC flag is clear, the bootstrap attempt fails, and the subsequent console action is determined by Figure 27-1.

The console sets the BIP flag and clears the RC flag before transferring control to system software. System software sets the RC flag to indicate that sufficient context has been established to handle a restart attempt. System software clears the BIP flag to indicate that the bootstrap operation has been completed. The RC flag should be set before clearing the BIP flag. Table 27–8 gives the console interpretation of BIP and RC flags.

Table 27–8: Console Interpretation of BIP and RC Flags

BIP	RC	Interpretation at Entry to Console I/O Mode
set	clear	Failed bootstrap
set	set	Halt condition encountered during bootstrap, restart processor
clear	clear	Failed restart
clear	set	Halt condition encountered, restart processor

27.4.1.5 Loading of System Software

The console is responsible for loading system software at the base of Region 1 beginning at virtual address 512MB. This software is expected to be a primary bootstrap program that is responsible for loading other system software, but may be diagnostic or other special-purpose software. Section 27.6 contains descriptions of the format of each supported bootstrap medium.

The console uses the `BOOT_DEV` environment variable to determine the bootstrap device and the path to that device. Environment variables contain lists of bootstrap devices and paths; each list element specifies the complete path to a given bootstrap device. If multiple elements are specified, the console attempts to load a bootstrap image from each in turn.

The console uses the `BOOTDEF_DEV`, `BOOT_DEV`, and `BOOTED_DEV` environment variables as follows:

- At console initialization, the console sets the `BOOTDEF_DEV` and `BOOT_DEV` environment variables to be equivalent. The format of these environment variables depends on the console implementation and is independent of the console presentation layer; the value may be interpreted and modified by system software.
- When a bootstrap results from a `BOOT` command that specifies a bootstrap device list, the console uses the list specified with the command. The console modifies `BOOT_DEV` to contain the specified device list.

Note:

This may require conversion from the presentation layer format to the registered format.

- When a bootstrap is the result of a `BOOT` command that does not specify a bootstrap device list, the console uses the bootstrap device list contained in the `BOOTDEF_DEV` environment variable. The console copies the value of `BOOTDEF_DEV` to `BOOT_DEV`.
- When a bootstrap is not the result of a `BOOT` command, the console uses the bootstrap device list contained in the `BOOT_DEV` environment variable. The console does not modify the contents of `BOOT_DEV`.

- The console attempts to load a bootstrap image from each element of the bootstrap device list. If the list is exhausted before successfully transferring control to system software, the bootstrap attempt fails and the subsequent console action is determined by Figure 27–1.
- The console indicates the actual bootstrap path and device used in the BOOTED_DEV environment variable. The console sets BOOTED_DEV after loading the primary bootstrap image and before transferring control to system software. The BOOTED_DEV format follows that of a BOOT_DEV list element.
- If the bootstrap device list is empty, BOOTDEF_DEV or BOOT_DEV are NULL (00₁₆), and the action is implementation specific. The console may remain in console I/O mode or attempt to locate a bootstrap device in an implementation-specific manner.

The BOOT_FILE and BOOT_OSFLAGS environment variables are used as default values for the bootstrap file name and option flags. The console indicates the actual bootstrap image file name (if any) and option flags for the current bootstrap attempt in BOOTED_FILE and BOOTED_OSFLAGS and environment variables. The BOOT_FILE default bootstrap image file name is used whenever the bootstrap requires a file name and either none was specified on the BOOT command or the bootstrap was initiated by the console as the result of a major state transition. The console never interprets the bootstrap option flags, but simply passes them between the console presentation layer and system software.

27.4.1.6 Processor Initialization

Before control is transferred to system software, certain IPRs and other processor state must be initialized as shown in Table 27–9 and Section 27.3.2.3 for each PALcode variant. Processor initialization is performed by the console before booting a processor, before restarting a processor, or as the result of the INITIALIZE –CPU console command.

The Context Valid (CV) flag in the processor’s per-CPU slot must be valid for processor initialization to be successful. If the CV flag is clear, the HWPCB contained in the per-CPU slot is not valid, and the console must not transfer control to system software. If this or any error occurs in initializing the processor, the console retains control of the system and generates the binary error message ERROR_PROC_INIT.

Table 27–9 Processor Initialization

Processor State		Initialized State
ASN	Address Space Number	Zero
ASTEN ¹	AST Enable	ASTEN in processor’s HWPCB
ASTSR ¹	AST Summary	ASTSR in processor’s HWPCB
BIP and RC flags		Unaffected
Cache, instruction buffer, or write buffer		Empty or valid
Environment variables		Unaffected
FEN	Floating Enable	FEN in processor’s HWPCB
Halt Data Log PA		Physical address of in-memory buffer of console data to be passed to the operating system (if any, otherwise 0).

Table 27–9 Processor Initialization (Continued)

Processor State		Initialized State
Halt Data Log Length		Length in bytes of console data (if any, otherwise 0).
Integer and floating-point registers		Unaffected, except SP
IPL	Interrupt Priority Level	Highest
Main memory		Unaffected
MCES	Machine Check Error Summary	8 (bit 3=1)
Other HWRPB fields		Unaffected
Other IPRs		UNPREDICTABLE
PCBB	Privileged Context Block	Address of processor's HWPCB
PS	Processor Status	IPL=highest, VMM=0, CM=K, SW=0
PTBR	Page Table Base Register	PFN value in processor's HWPCB
Reason for Halt code		Unaffected
SCC ¹	System Cycle Counter	Zero
SISR ¹	Software Interrupt Summary	Zero
SP	Kernel Stack Pointer	KSP in processor's HWPCB
Translation buffer		Invalidated
VIRBND	Virtual Boundary Register	-1
WHAMI	Who-Am-I	CPU identifier

¹ OpenVMS only.

27.4.1.7 Transfer of Control to System Software

Before transferring control to system software, the console must define valid hardware privileged context for that software. The console builds that context in the hardware privileged context block (HWPCB) in the primary processor's per-CPU slot. The initialized context is summarized in Table 27–10 and Section 27.3.2.3 for each PALcode variant.

The initial KSP points to the lowest addressed quadword in the higher addressed stack guard page (top-of-stack) of Region 1 of the bootstrap address space. The PTBR points to the Level 1 page table page. All other scalar and floating-point register contents are UNPREDICTABLE.

After building the HWPCB for the primary processor, the console sets the Context Valid (CV) flag in the primary's per-CPU slot. All other bootstrap information is passed from the console to system software by environment variables. See Section 26.2 for more details.

Table 27–10: Initial HWPCB Contents

HWPCB Field	Initialized State
KSP	Top-of-stack (contents of SP)
ESP ¹	UNPREDICTABLE
SSP ¹	UNPREDICTABLE
USP	UNPREDICTABLE
PTBR	PFN of Level 1 page table
ASN	Zero
ASTSR ¹	Zero
ASTEN ¹	Zero (all disabled)
FEN	Zero (disabled)
PCC	Zero
Unique value	Zero
PALcode scratch	Implementation specific

¹ OpenVMS systems only.

Control is transferred to system software in kernel mode at the highest IPL with virtual memory management enabled. Control is transferred to the first longword of the system software image loaded into Region 1, virtual address 0000 0000 2000 0000₁₆. Before transferring control, the console ensures that the SP contains the KSP value in the HWPCB. System software should assume that the stack is initially empty.

The transfer of control transitions the primary processor from the halted state into the running state and from console I/O mode into program I/O mode. The rest of the uniprocessor bootstrap process is the responsibility of system software.

27.4.2 Warm Bootstrapping in a Uniprocessor Environment

The actions of the console on a warm bootstrap are a subset of those for a cold bootstrap. A system bootstrap will be a warm bootstrap whenever the BOOT_RESET environment variable is set to "OFF", and console internal state permits.

The console performs the following steps in the warm bootstrap sequence:

1. Locate and validate the Hardware Restart Parameter Block (HWRPB)
2. Locate and load the system software primary bootstrap image
3. Initialize processor state on all processors
4. Initialize bootstrap page tables and map initial regions

5. Transfer control to the system software primary bootstrap image

At warm bootstrap, the console does not load PALcode, does not modify the Memory Data Descriptors, and does not reinitialize any environment variables. If the console cannot locate and validate the previously initialized HWRPB and Memory Cluster Descriptors, the console must initiate a cold bootstrap. Before beginning a bootstrap, the console must clear any internally pending restarts to any processor.

Programming Note:

Warm bootstrap permits system software to preserve limited context across bootstraps.

27.4.2.1 HWRPB Location and Validation

After console initialization, the console must preserve the location of the HWRPB in an implementation-specific manner. On warm bootstraps and restarts, the console locates the HWRPB and verifies it by ensuring that:

1. The first quadword of the table contains the physical address of the table.
2. The second quadword of the table contains "HWRPB" (0000 0042 5052 5748₁₆).
3. The quadword at offset HWRPB[288] contains the 64-bit sum, ignoring overflows of the quadwords from offset HWRPB[00] to HWRPB[280], inclusive, relative to the beginning of the potential HWRPB.
4. The quadword at offset [0] of the MEMDSC block contains the 64-bit sum, ignoring overflows, of the quadwords from MEMDSC+8 through MEMDSC_END of that block. The MEMDSC block is located by the MEMDSC offset at HWRPB[200]. See Figure 27–2.
5. As of HWRPB Revision 12, the type of memory cluster descriptors embedded within the MEMDSC block is checked. If the MEMDSC block contains static memory cluster descriptors (Figure 3-3), then skip this step.

If the MEMDSC block contains a null memory cluster descriptor (Figure 27–4), then instance-private distributed memory cluster descriptors exist in the FRU table that must be verified, as follows:

- a. The FRU table's physical address (FRU_ADDR) is calculated by adding the FRU Table Offset at HWRPB[216] to the base physical address of the HWRPB.
- b. The physical address of the first memory descriptor (MCD_ADDR) is the sum of FRU_ADDR and the offset value found in MEMDSC[56]. See Figure 27–4.
- c. The quadword at offset [0] of the memory descriptor at MCD_ADDR contains the 64-bit sum, ignoring overflows, of the quadwords from DMEMC+8 through DMEMC_END of that descriptor. See Figure 27–5.
- d. The physical offset to the next memory descriptor is found at MCD_ADDR[8]. If that offset is not –1, add it to FRU_ADDR to calculate a new MCD_ADDR and go to Step C.

If one or more of the above conditions is not true, the HWRPB is not valid. The warm bootstrap (or restart) fails. The subsequent console action is determined by Figure 27–1. If a bootstrap is indicated, a cold bootstrap will be performed.

The console must not search memory for a HWRPB; searching memory constitutes a security hole.

27.4.3 Multiprocessor Bootstrapping

Multiprocessor bootstrapping differs from uniprocessor bootstrapping primarily in areas relating to synchronization between processors. In a shared memory system, processors cannot independently load and start system software; bootstrapping is controlled by the primary processor.

27.4.3.1 Selection of Primary Processor

The primary processor is selected by the console during system initialization before any access to main memory by any processor. Selection of the primary processor may be done in any fashion that guarantees choosing exactly one primary processor.

Once a primary processor has been selected, the secondary processors take no further action until appropriately notified by the primary processor. In particular, secondary processors must not access main memory.

27.4.3.2 Actions of Console

After selection, the console proceeds to bootstrap the primary processor, after the normal uniprocessor bootstrap as described in Section 27.4.1.

The console must correctly initialize all HWRPB fields used for synchronization or communication between the processors. The console must initialize the PRIMARY CPU ID field at HWRPB[32], zero the TXRDY and RXRDY bitmasks (see Section 26.4), and recompute the HWRPB checksum at HWRPB[288].

The console must also initialize each per-CPU slot for the secondary processors, even for those processors that are not yet present. The console must:

- Clear the BIP, RC, OH, and CV flags
- Clear the Halt Request code field
- Set the PP flag if the processor is present
- Set the PA flag if the processor is present and available for use by system software
- Set the PMV and PL flags if the console has loaded PALcode on this processor
- Set the PV flag if the console has initialized PALcode on this processor
- Set the PE processor variation flag if the processor is eligible to become a primary
- Set the Console Data Log physical address entry to the physical address of the Console Data Log (if any, otherwise 0).
- Set the Console Data Log length to the length in bytes of the Console Data Log (if any, otherwise 0).

After initializing each processor's per-CPU slot, the console must notify each console secondary processor of the existence and location of the valid HWRPB.

27.4.3.3 PALcode Loading on Secondary Processors

Most console implementations load PALcode on all secondary processors before bootstrapping the primary processor. Console implementations may delay the loading or initialization of PALcode on a secondary processor. If delayed, PALcode loading and initialization require the cooperation of system software executing on the running primary and the console executing on behalf of the secondary.

The console secondary must have performed any necessary initialization as described in Section 27.4.3.5. All interprocessor console communications follow the mechanisms described in Section 26.4.

The following procedure applies only to initial PALcode loading on a console secondary. The PALcode variant to be loaded must be identical to that of the running primary processor before any PALcode switching by system software. This procedure cannot be used to load operating system-specific PALcode variants:

1. The console secondary initializes the PALcode memory and scratch space length fields in its per-CPU slot.
2. The console secondary sets the PALcode major revision, minor revision, and compatibility subfields in the PALcode revision field in its per-CPU slot.
3. The console secondary notifies the primary that PALcode loading is requested by transmitting a message to the running primary as described in Section 26.4.
4. The console secondary polls the PALcode Memory Valid (PMV) flag in its per-CPU slot.
5. The running primary detects the console secondary request.
6. The running primary verifies that the Processor Available (PA) flag is set in the secondary's per-CPU slot. If the flag is not set, the operation fails.
7. The running primary compares the major and minor revision subfields of the PALcode revision field in its per-CPU slot to that in the secondary's per-CPU slot. If the revision levels do not match, the running primary proceeds to step 12.
8. The running primary compares the number of processors currently sharing its PALcode image to the maximum contained in the subfield of the PALcode revision field of its per-CPU slot. If the current number is the maximum, no additional console secondary can share the PALcode image. The running primary proceeds to step 12.

Programming Note:

The running primary can determine the number of processors currently sharing a given PALcode image by counting the number of per-CPU slots with the same valid PALcode memory space descriptors. A PALcode memory space descriptor is valid if the PALcode Loaded (PL) flag is set in the per-CPU slot.

9. The running primary copies the PALcode memory and scratch space descriptors from its per-CPU slot into the secondary's per-CPU slot.
10. The running primary copies the PALcode variation, compatibility, and maximum number of processors subfields of the PALcode revision field from its per-CPU slot into the secondary's per-CPU slot.
11. The running primary sets the PALcode Loaded (PL) flag in the secondary's per-CPU slot, then proceeds to step 13.

12. The running primary allocates physical memory for PALcode memory and scratch areas and records the addresses in the secondary's per-CPU slot.
13. The running primary sets the PALcode Memory Valid (PMV) flag in the secondary's per-CPU slot.
14. The console secondary observes that the PMV flag is set in its per-CPU slot.
15. If the PL flag in its per-CPU slot is not set, the console secondary loads PALcode into the allocated PALcode memory and scratch space. In this case, the console secondary sets the PALcode Loaded (PL) flag in its per-CPU slot.
16. The console secondary ensures that any required implementation-specific PALcode initialization is performed.
17. The console secondary sets the PALcode Valid (PV) flag in the secondary's per-CPU slot.

The PALcode memory and scratch space must be page aligned. If not allocated by the console before system bootstrap, the allocation management of PALcode memory for secondary processors is the responsibility of system software.

It is the responsibility of console and system software to ensure that the initially loaded PALcode variation and revision levels of all processors are compatible. This may be performed by the primary before starting the secondary, by the starting secondary, or any combination thereof. PALcode images of the same PALcode variation but different revision levels are compatible if the PALcode revision compatibility subfields match.

27.4.3.4 Actions of the Running Primary

System software executing on the primary processor must initialize the HWPCB for each secondary processor. The HWPCB contains the necessary privileged context for the execution of system software and successful restarts. The HWPCB must be initialized before requesting that the console secondary perform any START command. After initializing the HWPCB, system software sets the Context Valid (CV) flag.

Once the PALcode is valid on a console secondary, the secondary waits for a START (or other) command from the running primary. System software issues the necessary console commands that instruct the secondary to begin executing software. The exchange of commands and messages between the running primary and a secondary is described in Section 26.4.

System software may start secondary processors at any time. In particular, secondary processors may be started before or after switching PALcode on the running primary. If system software switches to an operating system-specific PALcode before starting a secondary processor, system software must update the PALcode revision field in the per-CPU slot (SLOT[168]) of each secondary before starting the secondary. See Section 27.3.1.

Programming Note:

All commands sent to a console secondary are implicitly targeted to the secondary.

27.4.3.5 Actions of a Console Secondary

After failing to become the primary, a console secondary uses an implementation-specific mechanism to determine when a valid HWRPB has been constructed in main memory. The console secondary then locates the HWRPB in an implementation-specific manner.

Once the HWRPB is located, the secondary locates its per-CPU slot using its CPU ID as an index. The secondary verifies that its slot exists by comparing its CPU ID to the number of per-CPU slots at HWRPB[144]. If its CPU ID exceeds the number of per-CPU slots, the secondary must not leave console mode or continue to access main memory. If PALcode loading is necessary, the console secondary follows the procedure given in Section 27.4.3.3.

Once PALcode is valid, the console secondary waits for a START (or other) command from the running primary by polling the appropriate flag in the RXRDY bitmask. The exchange of commands and messages between the running primary and a secondary is described in Section 26.4.

In response to a START command, the console secondary:

1. Verifies that the Context Valid (CV) flag is set in its per-CPU slot.
2. Sets the Bootstrap-in-Progress (BIP) flag in its per-CPU slot.
3. Clears the Restart-Capable (RC) flag in its per-CPU slot.
4. Initializes the processor (see Table 27–9).
5. Set the Cache Information fields in its per-CPU slot.
6. Set the Console Data Log PA and Length fields in its per-CPU slot (zero if none).
7. If necessary, switches to the system software specific PALcode variant identified in the PALcode revision field in its per-CPU slot.
8. Loads the privileged context specified by the HWPCB in its per-CPU slot.
9. Loads the procedure value at HWRPB[264] into R27.
10. Clears R26 and R25.
11. Loads the virtual page table base (VPTB) register with the value stored in HWRPB[120].
12. Transfers control to the CPU Restart routine, whose virtual address is stored in HWRPB[256].

The CV flag indicates that the HWPCB in the slot contains valid hardware privileged state for system software. If the CV flag is not set, the processor remains in console I/O mode.

The console uses the PALcode revision field in the per-CPU slot to determine if system software has switched PALcode to a system software-specific variant. The console must restore that variant before passing control to the CPU restart routine.

27.4.3.6 Bootstrap Flags

The Bootstrap-in-Progress (BIP) and Restart-Capable (RC) processor state flags in the console secondary processor's per-CPU slot are used to control error recovery during secondary starts. If the secondary re-enters console I/O mode while the BIP flag is set and the RC flag is clear, the start attempt fails. Failed starts are equivalent to failed bootstraps, and the subsequent console action is determined by Figure 27–1. See Section 27.4.1.3 and Table 27–8.

27.4.4 Addition of a Processor to a Running System

A processor may be added to a running system at any time if a slot has been provided for it in the HWRPB. The new console secondary processor follows the secondary start procedure given in Sections 27.4.3.3 and 27.4.3.5, with one minor difference. If no PALcode loading is

necessary, the console secondary sends a ?STARTREQ? message to the running primary. This message notifies the primary that a new processor has been added to the configuration. After sending the ?STARTREQ? message, the console secondary waits for a START (or other) command from the running primary. See Section 26.4 for a description of interprocessor console communication.

27.4.5 System Software Requested Bootstraps

System software can request that the console perform a system bootstrap. This request can be made on any processor in a multiprocessor system and overrides the setting of the AUTO_ACTION and BOOT_RESET environment variables.

To request a bootstrap, system software sets one of the codes requested by the bootstrap in the Halt Request field of its per-CPU slot, then executes a CALL_PAL HALT instruction. If a cold bootstrap is requested, the "Cold Bootstrap Requested" code ('2') is set; the "Warm Bootstrap Requested" ('3') code is set to request a warm bootstrap.

Instead of initiating the normal error halt processing described in Section 27.5.4, the console initiates the appropriate system bootstrap as described in Sections 27.4.1 and 27.4.2. The bootstrap attempt is unconditional; the AUTO_ACTION or the BOOT_RESET environment variables do not affect the bootstrap attempt.

27.5 System Restarts

The console is responsible for restarting a processor halted by powerfail or by error halt. The console follows the same sequence for a primary or secondary processor.

27.5.1 Actions of Console

The console begins the restart sequence by locating and then validating the HWRPB, using the procedure given in Section 27.4.2.1. If the HWRPB is not valid, the restart attempt fails. See Section 27.1.1 for console actions at major state transitions.

If the HWRPB is valid, the console uses the processor CPU ID as an index to calculate the address of that processor's HWRPB slot. The console:

1. Verifies that the processor's PALcode Valid (PV) flag is set. If the PV flag is clear, PALcode is not valid, and the restart attempt fails.
2. Verifies that the processor's Context Valid (CV) flag is set. If the CV flag is clear, the HWPCB does not contain valid software context for the restart, and the restart attempt fails.
3. If the Reason for Halt is anything other than "powerfail restart", the console examines the processor's Restart-Capable (RC) flag. If RC is set, the console proceeds with the restart at step 5. If RC is clear, system software is not capable of attempting the restart, and the restart attempt fails.

Ignoring the RC flag for powerfail restart avoids unnecessary bootstraps that are caused by repeated power failures that in turn, are caused by a bouncing power supply that prevents software from having sufficient time to set the RC flag.

4. Examines the Bootstrap-in-Progress (BIP) flag. If BIP is clear, and the AUTO_ACTION environment variable is "BOOT", a system bootstrap is attempted. Otherwise, the processor remains in console I/O mode. See Figure 27–1.
5. Examines the PALcode revision field in its per-CPU slot. If the revision field does not match the PALcode revision in use by the console, the console must switch PALcode before passing control to the CPU Restart routine.
6. Loads the privileged context specified by the HWPCB in its per-CPU slot.
7. Loads the procedure value at HWRPB[264] into R27.
8. Clears R26 (return address) and R25 (argument information).
9. Loads the virtual page table base (VPTB) register with the value stored in HWRPB[120].
10. Transfers control to the CPU Restart routine, whose virtual address is stored in HWRPB[256].

On all restart attempt failures the console initiates the action indicated by Figure 27–1. The PV and CV flags should never be clear for the primary processor; if either flag is clear, then the restart fails. Also, no PALcode or system software is loaded during a restart.

It is the responsibility of system software to complete the restart operation and to set the RC flag at the point where a subsequent restart can be handled correctly.

27.5.2 Powerfail and Recovery — Uniprocessor

On Alpha systems, the system power supply conditions external power and transforms it for use by the processor, memory, and I/O subsystems. Backup options are available on some systems to supply power after external power fails. The backup option may supply power to all of the system platform hardware or only a subset. The effect of an external power failure depends on the backup option:

- If no backup option exists, the processor cannot be restarted after power is restored. The processor must be bootstrapped or left halted in console I/O mode.
- If the backup option maintains power to all of the system platform hardware, execution of system software is unaffected by the power failure. It must be possible for system software to determine that a transition to backup power has occurred.
- If the backup option maintains only the contents of memory and keeps system time with the BB_WATCH, the power supply must request a powerfail interrupt. After requesting the interrupt, the power supply must continue to supply power to the processor for an implementation-specific period to allow system software to save state.

Powerfail recovery is possible only if adequate system state is preserved during an interruption of power to the processor. System software must save all volatile state and perform any operating system-specific actions necessary to ensure later successful recovery.

When power is restored, the console determines that the HWRPB is still valid, then examines the console lock and AUTO_ACTION environment variable. If the console is locked, and AUTO_ACTION environment variable is "RESTART", the console attempts an operating system restart. See Section 27.1.1.

The processor may lose state when power is lost. For example, if a processor is halted when power fails, the action on power-up is still determined by the console switches and environment variables. The system does not necessarily stay halted.

Software Note:

As explained in Chapter 14 for OpenVMS and Chapters 19 and 24 for Tru64 UNIX and Alpha Linux, respectively, a powerfail interrupt is delivered at an appropriate IPL to the interrupt service routine located at SCB offset 640_{16} for that operating system.

27.5.3 Powerfail and Recovery — Multiprocessor

There are two basic approaches to powerfail recovery on multiprocessor systems:

- **United**
All available processors effectively experience the powerfail event identically.
- **Split**
Each available processor effectively experiences independent powerfail events.

A processor is "available" if the Processor Available (PA) flag is set in the processor's per-CPU slot. The powerfail system variation flag at HWRPB[88] indicates the type of powerfail and restart action.

A multiprocessor Alpha system that supports powerfail recovery must implement the united powerfail mode. The split mode may be implemented optionally as an alternative, selected at system bootstrap.

Software Note:

OpenVMS supports only the united powerfail and recovery mode at this time. Powerfail recovery is possible only when the primary is restarted; all secondaries should remain in console I/O mode.

27.5.3.1 United Powerfail and Recovery

In united powerfail and recovery mode, all available processors experience powerfail interrupts, halts, and restorations uniformly. If one available processor experiences a powerfail event, all other available processors experience that event. Therefore, if one processor powerfails and recovers, all processors must do so. Even if a separately powered processor does not actually lose power, that processor will still receive the powerfail interrupt and must be restarted as if power had been lost.

When power is restored and a restart is to be attempted, the console must determine whether to restart all available processors or only the primary processor. The console determines the appropriate action by the Powerfail Restart (PR) flag in the system variation field of the HWRPB[88]. If the PR flag is set, the console attempts to restart all available processors; if PR is clear, the console attempts to restart only the primary processor. In both cases, it is the responsibility of system software to coordinate and synchronize further powerfail recovery.

27.5.3.2 Split Powerfail and Recovery

In split powerfail and recovery mode, only the available processors that actually experience a loss of power will experience a powerfail interrupt and subsequent recovery. Available processors that are separately powered and do not lose power do not experience a powerfail interrupt.

When power is restored and a restart is to be attempted, the console must determine whether to restart any available processor or only the primary processor. As in the united mode, the console determines the appropriate action by the Powerfail Restart (PR) flag in the system variation field of the HWRPB[88]. If the PR flag is set, the console attempts to restart any available processor. If PR is clear, the console attempts to restart only the primary processor; on a secondary, the console sends the ?STARTREQ? message and waits for a START (or other command) from the running primary as discussed in Section 27.4.3.5. Again, system software has the responsibility for further coordination and synchronization of powerfail recovery.

27.5.4 Error Halt and Recovery

A number of serious error conditions can prevent a processor from executing the current thread of software. Such error conditions are detected by PALcode and halt the processor.

When a halt is encountered, the console must ensure that the processor hardware state is visible to the console operator and to system software after a subsequent restart attempt. This state includes the current values in PS, PC, SP, PCBB, HWPCB, all integer registers, all floating-point registers, and the name of the halt condition. The console must:

1. Ensure that the contents of the integer and floating-point registers appear unaffected.
2. Write the current hardware context to the HWPCB located by the current PCBB.
3. Write the current PS, PC, and PCBB register contents into the processor's per-CPU slot.
4. Write the current R25, R26, and R27 register contents into the processor's per-CPU slot.
5. Set the appropriate code into the Reason for Halt field of the processor's per-CPU slot.

The values of R25, R26, and R27 must be explicitly saved in the per-CPU slot to permit the console to invoke the CPU restart routine.

Section 27.1.1 and Table 26–4 list the defined halt conditions that transition an Alpha processor from the running state to a halted state and that may lead to an attempt to restart the processor. Each condition is passed to the operating system in the Reason for Halt quadword of the processor's HWRPB slot.

When an error halt occurs, the console examines the console lock setting. If the console is locked, the console attempts a restart. If unlocked, the console action is determined by the setting of the AUTO_ACTION environment variable (see Figure 27–1). See Section 27.5.1 for a description of the restart attempt process.

The processor must be initialized after an error halt. If the processor starts running after an error halt without an intervening processor initialization, the operation of the processor is UNDEFINED. The effects of processor initialization are summarized in Table 27–9.

An error halt directly affects only the processor that incurred it, although multiple processors may simultaneously and coincidentally incur their own error halt conditions. If restarts are enabled, each halted processor must be independently restarted by the console. The restarts of individual processors may occur in a different order than the error halts occurred, but if the

console restarts any halted processor, it must restart all halted processors in a timely fashion unless a bootstrap is requested in the meantime. A bootstrap nullifies any pending restarts in the multiprocessor.

27.5.5 Operator Requested Crash

When the operating system does not respond to normal program requests, the console operator may request that the console request an operating system crash. A console requested crash differs from a console halt of a processor in that system software can write a crash dump.

The console operator interacts with the console presentation layer and requests the crash with a HALT –CRASH command. The console converts this command to an error halt restart of system software. After gaining control of the processor, the console preserves the hardware state (see Section 27.5.4). The console passes the crash request to system software by using the "Console Operator requests system crash" code in the Reason for Halt field in the primary's per-CPU slot. It is the responsibility of the system software restart routine to initiate the crash in an implementation-specific fashion.

27.5.6 Primary Switching

System software may find it necessary to replace the primary processor with one of the running secondary processors without bootstrapping the system. This "switch" of the running primary may be caused by an error encountered by the primary or by a program request. Switching a running primary must be initiated by system software; the console cannot force a switch to occur.

Support for primary switching is optional to system software, console implementations, and system platforms. The system platform hardware must permit the selected secondary to assume the functions of a primary. The selected secondary must have direct access to the console, a BB_WATCH, and all I/O devices. Direct access to the console ensures that the secondary can access console I/O devices and the console terminal. Direct access to a BB_WATCH ensures that the secondary can act as the system timekeeper. Direct access to all I/O devices ensures that the secondary can initiate I/O requests to and receive I/O interrupts from all I/O devices, and that the secondary can reinitialize all devices as part of powerfail recovery.

If the processor is eligible to become a primary, the console will set the Primary Eligible (PE) processor variation flag in the processor's per-CPU slot during processor initialization. See Table 26–4.

Primary switching requires cooperation between system software and the console. System software is responsible for the selection of the new primary and any necessary redirection of I/O interrupts. The console is responsible for any necessary configuration of the console terminal or other console device interface.

27.5.6.1 Sequence on an Embedded Console

The sequence of events differs depending on the type of console implementation. On a system with an embedded console, the operation proceeds as follows:

1. System software performs any actions specific to system software synchronization.
2. System software executing on the old primary ensures that the console terminal is in a quiescent state. In particular, character reception from the terminal must be suspended.

3. System software selects the new primary. The selected secondary must be eligible as indicated by the PE processor variation flag in its per-CPU slot.
4. System software executing on the old primary invokes the PSWITCH console callback specifying the "transition from primary" action.
5. The console attempts to perform any necessary hardware state changes to transform the old primary into a secondary.

Hardware/Software Coordination Note:

An example of such a hardware state change is disabling a console UART physically located on the processor board.

6. If the state change is completed, PSWITCH returns success status. System software may proceed with the primary switch at step 8.
7. If the state change is not effected, PSWITCH returns failure status. System software must take other appropriate action.
8. System software executing on the old primary notifies system software on the selected secondary of the successful PSWITCH completion.
9. System software executing on the selected secondary invokes the PSWITCH console callback specifying the "transition to primary" action.
10. The console verifies that the selected secondary is eligible to become a primary and attempts to perform any necessary hardware state changes to transform the old secondary into the new primary.
11. If the state change is completed, PSWITCH returns success status. System software may proceed with the primary switch at step 13.
12. If the state change is not effected, PSWITCH returns failure status. System software must select a different potential primary or take other appropriate action.
13. System software executing on the selected secondary reactivates the console terminal. In particular, character reception from the terminal is re-enabled.
14. System software performs any additional system reconfiguration, updates the PRIMARY CPU ID field at HWRPB[32], recomputes the HWRPB checksum at HWRPB[288], and performs any actions specific to system software synchronization.

27.5.6.2 Sequence on a Detached Console

On a system with a detached console, the operation is similar, but only one call to PSWITCH is required. Additional calls to PSWITCH with the "switch primary" action may result in UNDEFINED operation. The operation proceeds as follows:

1. System software performs any actions specific to system software synchronization.
2. System software executing on the old primary ensures that the console terminal is in a quiescent state. In particular, character reception from the terminal must be suspended.
3. System software selects the new primary. The selected secondary must be eligible as indicated by the PE processor variation flag in its per-CPU slot.
4. System software executing on any processor invokes the PSWITCH console callback specifying the "switch primary" action and the CPU ID of the new primary.

5. The console verifies that the selected secondary is eligible to become a primary and attempts to perform any necessary hardware state changes to transform the old primary into a secondary and to transform the selected secondary into the primary.
6. If the state change is completed, PSWITCH returns success status. System software may proceed with the primary switch at step 9.
7. If the state change is not effected and the resulting hardware state permits a return to system software, PSWITCH returns failure status. System software must select a different potential primary or take other appropriate action.
8. If the state change is not effected and the resulting hardware state does not permit a return to system software, the console takes the action associated with a failed restart.
9. System software executing on the selected secondary reactivates the console terminal. In particular, character reception from the terminal is re-enabled.
10. System software performs any additional system reconfiguration, updates the PRIMARY CPU ID field at HWRPB[32], recomputes the HWRPB checksum at HWRPB[288], and performs any actions specific to system software synchronization.

27.5.7 Transitioning Console Terminal State During HALT/RESTART

Abrupt transitions from program I/O mode to console I/O mode may occur. Such transitions may be caused by execution of a CALL_PAL HALT instruction, a catastrophic error, or a console operator forcing the processor into console I/O mode. Upon transition to console I/O mode, the console must be able to regain control of the console terminal, even though system software may have changed the device characteristics.

The console may seize control of the console terminal without regard to system software when the transition is such that no return to program I/O mode is possible. Such transitions are normally associated with a catastrophic error.

If system software execution may be continued, the console must be able to restore the existing state of the console terminal. The console must regain and subsequently relinquish control of the console terminal with the cooperation of system software.

Hardware/Software Coordination Note:

This is particularly desirable on workstations when the console operator forces the processor into console I/O mode.

System software may provide SAVE_TERM and RESTORE_TERM routines that can be called by the console to save and restore the state of the console terminal. To provide these optional routines, system software loads the SAVE_TERM and RESTORE_TERM starting virtual address and procedure descriptor fields in the HWRPB and recomputes the HWRPB checksum at HWRPB[288]. At system bootstraps, the console sets these fields to zero.

The console calls SAVE_TERM and RESTORE_TERM in kernel mode at the highest IPL in the memory management policy established by system software. The console loads the routine procedure value into R27, clears R25 and R26, and then transfers control to system software at the starting virtual address. The procedure value and starting virtual address for SAVE_TERM are contained in HWRPB[224] and [232]; those for RESTORE_TERM are contained in HWRPB[240] and [248]. These routines are invoked only on the primary processor and only

upon an unexpected entry into console I/O mode. The console must preserve sufficient hardware state to permit the processor to be restarted before invoking these routines. See Section 27.5.4.

Exit from these routines must be accomplished by using the CALL_PAL HALT instruction to return the processor to console I/O mode; these routines do not use the RET subroutine return instruction. Before exiting, these routines must set the "SAVE_TERM/RESTORE_TERM exit" code ('1') in the Halt Request field of the primary's per-CPU slot and indicate success ('0') or failure ('1') status in R0<63>. The console will not attempt to continue system software if a failure status is returned.

SAVE_TERM and RESTORE_TERM may be called when system software has encountered an unexpected CALL_PAL HALT or other halt condition; system state may be corrupt. These routines must be written with few or no dependencies on possibly corrupt system state.

Hardware/Software Coordination Note:

A console terminal on a serial line may or may not have state that needs to be saved. A console terminal on a workstation may require the system software to "roll down" the current screen to expose the "console window" and "roll up" the "console window" to expose the current screen.

27.5.7.1 SAVE_TERM — Save Console Terminal State

Format:

status = SAVE_TERM

Inputs:

R27 = Procedure value (HWRPB[232])

Outputs:

status = R0; status:
R0<63> '0' Success, terminal state saved
'1' Failure, terminal state not saved
R0<62:0> SBZ

SAVE_TERM is called by the console after an unexpected entry to console mode. The routine performs any implementation-specific and device-specific actions necessary to save the state of the console terminal as established by system software. When the routine exits and console I/O mode is restored, the console is free to modify the existing console terminal state in any manner.

Table 27–11: Bootstrap Devices and Image Media (Continued)

Device Class	Data Link	Protocol
Local Tape	N/A	ANSI Bootblock
Network	NI, FDDI	MOP Bootp
ROM	N/A	ROM Bootblock

As explained in Section 27.4.1.4, the console attempts to load a bootstrap image from each element of a bootstrap device list until a successful image load is achieved. If the bootstrap image cannot be located or if the load fails for any reason, the console retains control of the system, generates the binary error message `AUDIT_BSTRAP_ABORT`, and then attempts to load a bootstrap image from the next bootstrap device list element. After a bootstrap image is successfully located and loaded, the console transfers control to system software as described in Section 27.4.

As the loading of the bootstrap image proceeds, the console optionally generates an audit trail of progress messages. The `ENABLE_AUDIT` environment variable controls audit trail generation. The audit trail begins with the `AUDIT_BOOT_STARTS` message. The audit trail continues with messages that are specific to the bootstrap device. Each consists of a binary message code that is interpreted by the console presentation layer.

27.6.1 Disk Bootstrapping

An Alpha primary bootstrap may be loaded from a directly accessed disk device. The console loads the "boot block" contained in the first logical block (LBN 0) of the disk. The boot block contains the starting logical block number (LBN) of the primary bootstrap program and the count of contiguous LBNs that make up that image.

The first 512 bytes of the boot block are structured as shown in Figure 27–9. The console loads the primary bootstrap without knowledge of the operating system file system. The boot block is (previously) initialized by the operating system. The actual size of a logical block is device-specific and may exceed 512 bytes.

Figure 27–9 Alpha Disk Boot Block

63	Reserved (VAX Compatibility)	0
	Reserved (Expansion)	:BB
	Reserved	:+136
	Count (LBNs)	:+472
	Starting LBN	:+480
	Flags	:+488
	Checksum	:+496
		:+504
		:+512

A local disk bootstrap proceeds as follows:

1. The console reads the boot block from LBN 0 of the specified disk device.

2. The console validates the boot block CHECKSUM; if the checksum is not validated, the bootstrap image load attempt aborts. The console computes the checksum of the first 63 quadwords in the block as a 64-bit sum, ignoring overflow. The computation includes both reserved regions. The computed checksum is compared to the CHECKSUM.
3. The console generates the AUDIT_CHECKSUM_GOOD message if the audit trail is enabled.
4. The console ensures that the FLAG quadword is zero; otherwise the bootstrap image load attempt aborts.
5. The console ensures that the COUNT is non-zero; otherwise the bootstrap image load attempt aborts. The count field indicates the number of contiguous logical blocks that contain the primary bootstrap.
6. The console generates the AUDIT_LOAD_BEGINS message if the audit trail is enabled.
7. The console reads the primary bootstrap image specified by COUNT and STARTING LBN into system memory; in any error occurs, the bootstrap image load attempt aborts.
The transfer begins at the logical block given by the STARTING LBN; a contiguous COUNT number of logical blocks is read. The image is read into a virtually contiguous system memory buffer; the starting virtual address is 0000 0000 2000 0000₁₆. (See Section 27.4.1.1.)

Errors include device hardware errors, the specified STARTING LBN not being present on the disk, or unexpectedly encountering the last logical block on the disk during the read.
8. The console generates the AUDIT_LOAD_DONE message when the load has completed; the message is generated only if the audit trail is enabled.
9. The console prepares to transfer control to the bootstrap program as described in Section 27.4.1.6.

Implementation Notes:

Unlike the VAX boot block support, *no* native Alpha code is contained in the boot block; the boot block contains *only* the LBN descriptor for the Alpha primary bootstrap image. An Alpha boot can contain pointers to primary bootstrap images for both VAX and Alpha simultaneously.

Because the boot block includes an LBN and block count, the console need have no knowledge of the operating system file system or on-disk structure.

The first 136 bytes of the boot block are currently used by the VAX disk boot block mechanism. The next 80 bytes are not currently used either by VAX or Alpha boot blocks. For future expansions, VAX boot blocks should expand towards higher addresses and Alpha boot blocks expand towards lower addresses; each region remains contiguous. These 216 bytes are ignored by the Alpha console except for the purposes of computing the boot block checksum.

The boot block FLAGS word is reserved for future expansion. Flag<0> is reserved to indicate a discontinuous bootstrap image; Flag <63:1> are reserved for future definition. There are no current plans by any Compaq operating system to have a discontinuous primary bootstrap image.

27.6.2 Tape Bootstrapping

An Alpha primary bootstrap may be loaded from a directly accessed tape device. Before loading the primary bootstrap, the console must determine the tape format and locate the primary bootstrap on the tape. The console:

1. Rewinds the tape on the specified tape device to the beginning of the tape (BOT).
2. Reads the first record.
3. Determines the record length.
 - If the record length is 80 bytes, the tape may be an ANSI-formatted tape. The console proceeds as described in Section 27.6.2.1.
 - If the record length is 512 bytes, the tape is "boot blocked." The console proceeds as described in Section 27.6.2.2.
 - If the length is other than 80 or 512 bytes, the bootstrap image load attempt aborts.

27.6.2.1 Bootstrapping from ANSI-Formatted Tape

Before loading the primary bootstrap image from an ANSI-formatted tape, the console must ensure that the format is valid. To verify that a given record contains a particular ANSI label, the console checks for the ASCII label name string at the beginning of the record. For example, a record containing a VOL1 label begins with the ASCII string "VOL1." All other record bytes are ignored when verifying the label.

A primary bootstrap image file name may be specified explicitly on a BOOT command or implicitly by the BOOT_FILE environment variable. If no file name is specified, the first file located will be used.

A local ANSI-formatted tape bootstrap proceeds as follows:

1. The console verifies that the first record contains a VOL1 label; if the verification fails, the bootstrap image load attempt aborts.
2. The console generates the AUDIT_TAPE_ANSI message if the audit trail is enabled.
3. If no file name was specified, the console advances the tape position to the End-of-Tape (EOT) side of the the first tape mark. The console proceeds to step 5.
4. If a file name was specified, the console attempts to locate that file on the tape. If the file cannot be located, the attempt to load the bootstrap image aborts. The console compares the specified file name with the file name present in each HDR1 label on the tape. At the first match, the console proceeds to step 5.

The console searches for the specified file, starting with the second tape record. The console reads 80-byte records from the tape until it encounters an HDR1 label, then proceeds as follows:

- a. The console generates the AUDIT_FILE_FOUND<filename> message, where <filename> is the value of the HDR1 label. The message is generated only if the audit trail is enabled.
- b. The console compares the specified file name with the 17-character File Identifier Field found in the HDR1 label.
- c. If a match occurs, the console advances the tape position to after the next tape mark and proceeds to step 5. (Any HDR2 or HDR3 labels are ignored.)

- d. If no match occurs, the console advances the tape position over the next three tape marks and reads the next record. If another tape mark is found, the logical end of volume has been encountered and the attempt to load the bootstrap image aborts. Otherwise, the record should be the HDR1 label for the next file on the tape and the console proceeds at step A.

The console aborts the attempt to load the bootstrap image whenever an unexpected tape mark is encountered, the tape runs off the end, or a hardware error occurs.

5. The console generates the `AUDIT_LOAD_BEGINS` message if the audit trail is enabled.
6. The console reads the primary bootstrap image from tape into system memory; if any error occurs or if the tape runs off the end, the attempt to load the bootstrap image aborts.

The transfer from tape begins at the current tape position and continues until a tape mark is encountered. The image is read into a virtually contiguous system memory buffer; the starting virtual address is `0000 0000 2000 000016`. (See Section 27.4.1.1.)

7. The console checks that the bootstrap file was properly closed by:
 - a. Reading the record after the tape mark and verifying that the record is an EOF1 label. If not, the attempt to load the bootstrap image aborts.
 - b. Searching for a subsequent tape mark. If a tape mark is not found, the bootstrap file was improperly closed and the attempt to load the bootstrap image aborts. (Any EOF2 and EOF3 labels are ignored.)
8. The console generates the `AUDIT_LOAD_DONE` message if the audit trail is enabled.
9. The console prepares to transfer control to the bootstrap as described in Section 27.4.1.6. The console does not rewind or otherwise change the position of the tape after reading the bootstrap image.

27.6.2.2 Bootstrapping from Boot-Blocked Tape

Bootstrapping from a boot-blocked tape is similar to the local disk bootstrapping described in Section 27.6.1. The first tape record must be 512 bytes and must follow the format given for disk boot blocks as shown in Figure 27–9. The `STARTING LBN` and `FLAGS` fields are `MBZ` for tape boot boot blocks.

All tape records that comprise the primary bootstrap must be 512 bytes in size. If the console encounters records of any other size, the attempt to load the bootstrap image aborts.

A local tape boot block bootstrap proceeds as follows:

1. The console generates the `AUDIT_TAPE_BBLOCK` message if the audit trail is enabled.
2. The console validates the boot block `CHECKSUM`; if the checksum is not validated, the attempt to load the bootstrap image aborts. The console computes the checksum of the first 63 quadwords in the block as a 64-bit sum, ignoring overflow. The computation includes both reserved regions and the `MBZ` fields. The computed checksum is compared to the `CHECKSUM` at `[BB+504]`.
3. The console generates the `AUDIT_CHECKSUM_GOOD` message if the audit trail is enabled.

4. The console ensures that the COUNT is non-zero; otherwise the attempt to load the bootstrap image aborts. The count field indicates the number of subsequent 512-byte records that contain the primary bootstrap.
5. The console generates the AUDIT_LOAD_BEGINS message if the audit trail is enabled.
6. The console reads the count field subsequent records from the tape into system memory. The attempt to load the bootstrap image aborts if the console encounters any error, encounters any record size other than 512 bytes, or the tape runs off the end.

The image is read into a virtually contiguous system memory buffer; the starting virtual address is 0000 0000 2000 0000₁₆. (See Section 27.4.1.1.)

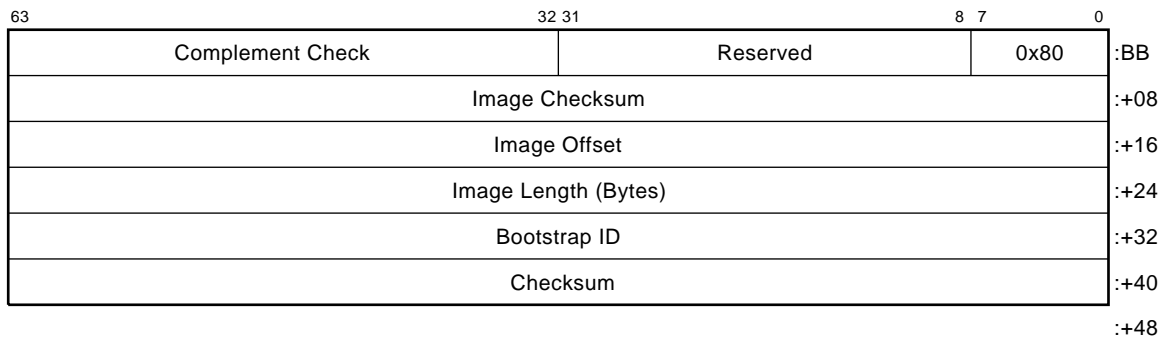
7. The console generates the AUDIT_LOAD_DONE message if the audit trail is enabled.
8. The console prepares to transfer control to the bootstrap as described in Section 27.4.1.6. The console does not rewind or otherwise change the position of the tape after reading the bootstrap image.

27.6.3 ROM Bootstrapping

An Alpha console may support bootstrapping from read-only memory (ROM). Bootstrap ROM is assumed to appear in multiple discontinuous regions of the physical address space. A given ROM region may contain multiple bootstrap images. A given bootstrap image must not span ROM regions.

Each ROM bootstrap image is page aligned and begins with a boot block as shown in Figure 27–10. The ROM boot block is similar to the local disk and tape boot block shown in Figure 27–9.

Figure 27–10 Alpha ROM Boot Block



A ROM bootstrap proceeds as follows:

1. The console locates the specified ordinal ROM bootstrap image; if the bootstrap image cannot be located, the attempt to load the bootstrap image aborts.

The console locates the ROM bootstrap image by searching ROM regions beginning with the ROM region with the lowest physical address and proceeding upward to the ROM region with the highest physical address.

The search proceeds as follows:

- a. The console verifies that the page contains a ROM bootstrap image:
 - The low-order byte of the first quadword must be 80₁₆.

- The high-order longword of the first quadword must be the one's complement of the low-order longword.
 - The sixth quadword must contain the checksum of the first five quadwords. The checksum is computed as a 64-bit sum, ignoring overflow.
- b. The console generates the `AUDIT_BOOT_TYPE<string>` message for each valid boot block, if the audit trail is enabled. The `<string>` is the ISO Latin-1 string contained in the `BOOTSTRAP ID` quadword.
 - c. If the specified ordinal image number has been reached, the console proceeds to step 2.
 - d. Otherwise, the console uses the `IMAGE LENGTH` at `[BB+24]` to determine the offset to the next ROM region page to be searched. The console repeats the process at step A.
2. The console computes the starting physical address of the bootstrap image by adding the physical address `OFFSET` at `[BB+16]` to the starting physical address of the boot block `[BB]`.
 3. The console verifies the accessibility of each page of the bootstrap image. If any page is inaccessible, the attempt to load the bootstrap image is aborted.
 4. The console generates the `AUDIT_BSTRAP_ACCESSIBLE` message if the audit trail is enabled.
 5. If requested, the console validates the `IMAGE CHECKSUM` at `[BB+08]`; if the checksum is not validated, the attempt to load the bootstrap image aborts. The console computes the checksum of all quadwords in the bootstrap image as a 64-bit sum, ignoring overflow. The existence and implementation of the mechanism for requesting this validation is implementation specific.
 6. The console generates the `AUDIT_BSTRAP_GOOD` message if the audit trail is enabled.
 7. If requested, the console copies the bootstrap image from ROM into system memory (RAM). The image is copied into a virtually contiguous buffer starting at virtual address `0000 0000 2000 000016`. (See Section 27.4.1.1.) The console generates the `AUDIT_LOAD_BEGINS` message before beginning the copy and the `AUDIT_LOAD_DONE` after the copy completes successfully if the audit trail is enabled.
 8. The console prepares to transfer control to the bootstrap as described in Section 27.4.1.6.

27.6.4 Network Bootstrapping

An Alpha system may support bootstrapping over one or more network communication devices and data link protocols. The console actions depend on the network device, data link protocol, and remote server capabilities.

An Alpha system can use the Compaq Network Architecture Maintenance Operations Protocol (MOP), or the BOOTP-UDP/IP network protocol, to bootstrap an Alpha system. See the MOP or BOOTP-UDP/IP specification for a detailed description.

A network bootstrap proceeds as follows:

1. The console determines if a bootstrap file name is to be used. The file name is taken from the BOOT command or the BOOT_FILE environment variable. If no file name is specified on the BOOT command and BOOT_FILE is null, no file name will be used.
2. The console generates the AUDIT_BOOT_REQ<filename> message if the audit trail is enabled.
3. The console issues the appropriate (MOP or BOOTP-UDP/IP) bootstrap request message(s).
4. The console receives an appropriate response (MOP or BOOTP-UDP/IP) from a remote bootstrap server. If no such response is received, the attempt to load the bootstrap image aborts.
5. The console generates the AUDIT_BSERVER_FOUND message if the audit trail is enabled.
6. The bootstrap load proceeds, using the appropriate network protocol.
7. When the console receives the first portion of the bootstrap image, the console generates the AUDIT_LOAD_BEGINS message if the audit trail is enabled.
8. The console loads the initial portion of the bootstrap image into a virtually contiguous system memory buffer; the starting virtual address is 0000 0000 2000 0000₁₆. (See Section 27.4.1.1.)
9. When the bootstrap image has been loaded, the console generates the AUDIT_LOAD_DONE message if the audit trail is enabled.
10. The console prepares to transfer control to the bootstrap program as described in Section 27.4.1.6.

If any error occurs, the attempt to load the bootstrap image aborts.

27.7 BB_WATCH

The following list offers important points about BB_WATCH:

1. BB_WATCH is the correct name for this entity. Although incorrect terminology, TOY, TODR, and watch chip, when used in the context of an Alpha system, are equivalent in meaning to the BB_WATCH.
2. System software must directly manipulate the BB_WATCH through an implementation-dependent interface.
3. System software makes the decision where to acquire known time; if a BB_WATCH is present, it may be used as the provider of known time.
4. Systems are not required to have a BB_WATCH.

Software Note:

However, all systems that support OpenVMS, Tru64 UNIX, and Alpha Linux must have a BB_WATCH.

5. If a BB_WATCH is present in a system, it meets the following requirements:
 - It has an accuracy of at least 50 ppm regardless of whether power is applied to the system.

- It has a resolution of at least 1 second (that is, it is read and written in units of a second or better).
 - Changing the entirety of the time maintained by the BB_WATCH takes under 1 second.
 - It has battery backup to survive a loss of power.
6. A BB_WATCH is always accessible to the primary processor. That is, a processor must be able to access a BB_WATCH directly (it must not need to go through another processor to access it) in order to be a candidate for primary processor.
 7. The number of BB_WATCH entities in a system is either one for the entire system or one for each processor in the system; which of the two options a system chooses is implementation dependent. If the latter option is chosen (one BB_WATCH per processor), writing one BB_WATCH does not update another.
 8. Although writing the BB_WATCH takes less than one second, it may not be a fast operation. Software should avoid frequently writing the BB_WATCH lest it negatively impact performance.
 9. The processor and its PALcode never changes the value of BB_WATCH except under the direction of system software. (The console, boot programs, and remote console clients are not system software.) The console, its PALcode, and any console application (including a diagnostic supervisor) never changes BB_WATCH except under the direction of the console operator — even when the CPU is halted, the processor is being initialized, or the BB_WATCH has an invalid time.

Programming Note:

The Primary-Eligible (PE) bit in the per-CPU slot of the HWRPB for each processor indicates, among other things, whether the CPU has access to a BB_WATCH. See Section 26.1.3.

The description of primary switching details the actions taken in a multiprocessor system, including the requirement for the primary processor to have access to the BB_WATCH.

27.8 Implementation Considerations

27.8.1 Embedded Console

In an embedded console implementation, the console executes on the same processor as the operating system. In such an implementation, the state transitions as experienced by the processor are more conceptual. For example, the processor acting as the console will be executing instructions when in the halted state. The processor may also field console I/O mode exceptions and interrupts.

An embedded console may be implemented as an extension of PALcode or as a distinct software entity. The console may execute from dedicated RAM or ROM on the processor or, after console initialization, may execute from main memory.

An embedded console implementation must include a mechanism by which the primary processor can be forced into console I/O mode from program I/O mode. This enables the console operator to gain control of the system regardless of the state of the system software. See Section 25.1 for recommended and required mechanisms.

27.8.1.1 Multiprocessor Considerations

In a multiprocessor system, selection of the primary processor occurs before any access to main memory by any of the processors. At system cold start, each of the processors will be executing in console I/O mode. The necessary memory for console execution must be independent of main memory; the console must be executing from dedicated console RAM or ROM and/or a suitably configured processor cache.

The selection of the console primary requires one or more hardware registers with state that is shared by all processors. One possible example is a mutex contained in a single-bit register accessed only with LDQ_L/STQ_C instructions. The primary successfully gains ownership of the mutex. Implementations should include mechanisms for operator override of the selection process and for recovery if the selection process fails.

Once a console primary has been selected, the console secondaries take no further action until appropriately notified by the primary. In particular, console secondaries must not access main memory. The console primary is responsible for building the HWRPB and any console-internal data structures (such as environment variables) for the secondaries. When these structures have been initialized, the console primary must be able to signal one or more of the secondaries by additional hardware register(s).

The console primary allocates a HWRPB in main memory, initializes it, and stores its physical address in an implementation-specific, nonvolatile manner. The console primary then indicates the presence of the HWRPB and its location to all secondaries by an implementation-specific mechanism.

On system restarts, the console primary identifies itself by comparing its WHAMI register contents with the Primary CPU ID value stored in the HWRPB.

When executing in console I/O mode, all processors must observe the same values of all console environment variables. The values of the AUTO_ACTION and BOOT_RESET environment variables are particularly important. After failing to become the console primary processor, a console secondary waits to be notified that a valid HWRPB exists. Upon such notification by the primary, the console secondaries use the address provided by the primary to locate the HWRPB. The primary may be in either program I/O mode or console I/O mode.

On cold bootstrap, a console secondary must not access main memory until notified by the primary that a valid HWRPB exists. Thus, there must exist a mechanism that is not based on main memory whereby the primary may signal each of the secondaries. On warm bootstrap or restart, a secondary processor must locate its per-CPU slot in the HWRPB and poll its RXRDY bit.

Console processors must locate the HWRPB without searching memory; such a search constitutes a security hole. One possible implementation is to use an environment variable or other shared console data structure. The address of the HWRPB must be nonvolatile across power failures in systems that support powerfail recovery.

Console implementations that support SAVE_ENV must be able to execute the routine simultaneously on each processor. System software use of SAVE_ENV requires care. System software must invoke SAVE_ENV on all available processors, but cannot ensure that the nonvolatile storage is updated on processors that are not available at the time of update. If mismatch occurs, the console uses the nonvolatile values preserved by the primary processor.

27.8.2 Detached Console

In a detached console implementation, the console executes on a separate and distinct hardware platform. A detached console may have cooperating special code that executes on one of the processors in the system configuration.

Detached console implementations should provide a keep-alive function. System software should be able to detect failures of the path between the system platform and the console. The mechanism may be a single dedicated signal or periodic message exchange. System software should be able to continue to execute if a keep-alive failure occurs, and restoration of the connection (or console state) should not cause a system crash or other major state transition. The console should buffer any messages if a keep-alive failure occurs until reconnection occurs.

Detached consoles may maintain a local console log. The logging device and format are implementation specific.

Appendixes

The following appendixes are included in the *Alpha Architecture Reference Manual*:

- Appendix A , Software Considerations
- Appendix B , IEEE Floating-Point Conformance
- Appendix C , Instruction Summary
- Appendix D , Registered System and Processor Identifiers
- Appendix E , Waivers and Implementation-Dependent Functionality
- Appendix F , Windows NT Software

Appendix A

Software Considerations

The typical audience for this appendix should think of the following documents as comprising a recommended set of documentation:

- This document
- *Compiler Writer's Guide for the Alpha 21264/21364* (if appropriate)
- The hardware reference manual (HRM) for the particular implementation of interest

The latter two are available at: <ftp.compaq.com/pub/products/alphaCPUdocs>.

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.

Because 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 provide an advantage 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 determined 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 using the Compaq Continuous Profiling Infrastructure (DCPI) with ProfileMe, as described in Sections E.2.3 through E.2.5.

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. It is difficult to get much speed advantage from paired transfers unless the code being executed has instruc-

tions and data appropriately aligned on aligned octaword boundaries. Because this is difficult to retrofit to old code, the following sections sometimes encourage "over-aligning" to octaword boundaries for 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. Because the Alpha architecture has many implementations, an exact cache design cannot be outlined here.

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.

Quadword I-fetch implementations should give first priority to executing aligned quadwords quickly. Octaword-fetch implementations 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.

Compilers should consider the following points when choosing a near-term and long-range strategy for branch target alignment:

- The 21064 issue-stalls on UNOP if Rx is busy (that is, an operation or load that has been issued and has not yet delivered a new value to Rx).
- FNOP cannot be used when the compiler is producing "INTEGER_ONLY" code. Therefore, on the 21064, UNOP may be a better choice than NOP in many cases.
- The 21164 generally performs better when UNOP is used to align branch targets. (The exception to this is any case where NOP or FNOP can improve performance in a specific implementation by preventing "splitting." Splitting occurs when at least one of a set of instructions sent to the issue stage is operand issue-stalled or destination issue-stalled. Splitting prevents the issue stage of the pipeline from emptying and the next set of instructions from being sent to the issue stage. It is an implementation-specific effect.)
- The 21264 and 21364 can use any NOP instruction to align branch targets, but UNOP is recommended to maintain backwards compatibility.
- OpenVMS, Tru64 UNIX, and Alpha Linux use R30 as the stack pointer. Utilities that symbolize instructions may chose to recognize only LDQ_U R31,0(R30) for UNOP, and compilers generate this as the preferred form.

A.2.2 Multiple Instruction Issue — Factor of 3

Alpha implementations 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.

In general, the following rules will give a good hardware-software match, but compilers should implement model-specific switches to generate code tuned more exactly to a specific implementation.

21064 and 21164

Pick one from column A and one from column B (but only a total of one load/store/branch per pair). Implementors of multiple-issue machines should give first priority to dual-issuing at least the following pairs, and second priority to multiple-issue of other combinations.

Column A	Column B
Integer Operate	Floating Operate
Floating Load/Store	Integer Load/Store
Floating Branch	Integer Branch
	BR/BSR/JSR

21264 and 21364

Because the 21264 and 21364 can rearrange instruction order to achieve maximum throughput, a table like that above is insufficient. Instead, look at *Compiler Writer's Guide for the 21264/21364* at ftp.compaq.com/pub/products/alphaCPUdocs.

A.2.3 Branch Prediction and Minimizing Branch-Taken — Factor of 3

With in-order instruction-issue Alpha implementations, an unexpected change in I-stream address results in at least 10 lost instruction times, with many more lost in out-of-order, speculative-issue implementations. "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 is slower than straight-line code.

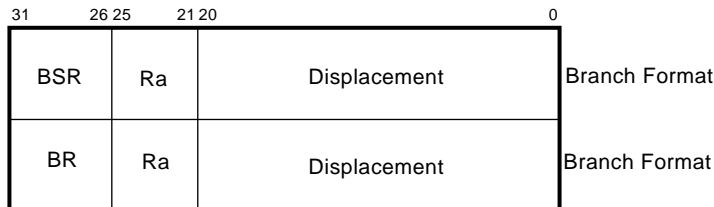
Compilers should follow these rules to minimize unexpected branches:

1. Branch prediction is implementation specific. 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 requires unrolling loops so that they contain at least 20 instructions, and putting subroutines of less than 20 instructions directly in line. It also requires 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 using the Conditional Move instructions can sometimes avoid breaking up basic blocks.
3. In an if-then-else construct for which the 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, as shown in Figure A-1.

Figure A-1: Branch-Format BSR and BR Opcodes



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, shown in Figure A-2, has 16 unused bits. These should be used by the compilers to communicate a hint about expected branch-target behavior (see Section 4.3.3).

Figure A-2: Memory-Format JSR Instruction



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 $(\text{updated PC} + \text{Instr} \langle 13:0 \rangle * 4) \langle 15:0 \rangle$ equals (likely_target_addr) <15:0>. In other words, pick the low 14 bits so that a normal $\text{PC} + \text{displacement} * 4$ 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 as quickly as possible with no branch-takens, 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 requires aligning branch targets and putting rarely executed code out of line.

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 writing a result register and an instruction using that register, if execution-time stalls are to be avoided. Thus, with a latency of zero, the instruction writes a result register and the instruction that uses that register can be multiple-issued in the *same* cycle. With a latency of 2, 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 unrolling loops and inlining short procedures.

Assume that all implementations can at least dual-issue instructions. Specific multiple instruction-issue rules and instruction latencies can be found in the hardware reference manual for the particular implementation. Scheduling techniques are located in the *Compiler Writer's Guide for the Alpha 21264/21364*. The manuals and guide are available at the following ftp site: <ftp.compaq.com/pub/products/alphaCPUdocs>.

Compilers should add implementation-specific switches to schedule code to match the 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 because 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. Give second priority 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 64 bytes will do well for most, if not all, Alpha implementations). This applies to array results that sweep through large portions of memory, and 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 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). 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.

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 affecting performance 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 are likely to be at least 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 results 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. Observing these guidelines 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 Conflicts — Factor of 1

Occasionally, programs that run with a direct-mapped cache will thrash, taking excessive cache 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 requires 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.

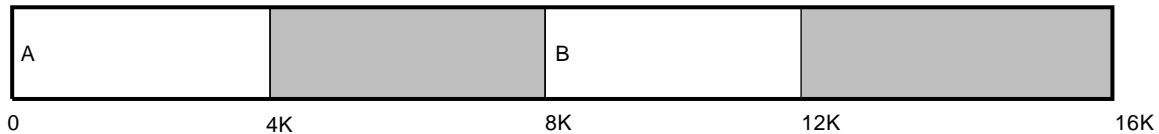
Example:

```
REAL*4 A(1000),B(1000)
DO 60 i=1,1000
60 A( i ) = f(B( i ))
```

Figures A-3, A-4, and A-5 show bad, better, and best allocation in cache, respectively.

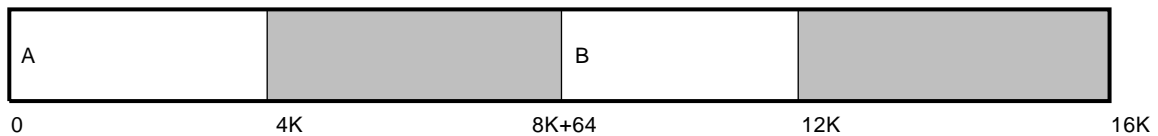
BAD allocation (A and B thrash in 8KB direct-mapped cache):

Figure A-3 Bad Allocation in Cache



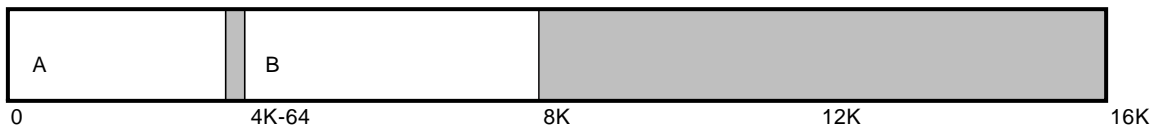
BETTER allocation (A and B offset by 64 mod 2KB, so 16 elements of A and 16 of B can be in cache simultaneously):

Figure A-4 Better Allocation in Cache



BEST allocation (A and B offset by 64 mod 2KB, so 16 elements of A and 16 of B can be in cache simultaneously, *and* both arrays fit entirely in 8KB or bigger cache):

Figure A-5 Best Allocation in Cache



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 8KB* page, or differ in bits VA<17:13>. For loops that go through arrays in a common direction with a common stride, this requires allocating the arrays, checking that the first-iteration addresses differ, and if they do not, inserting up to 8K bytes of padding *between* the arrays. This rule will avoid thrashing in some large direct-mapped data caches with total sizes of 32 pages (256KB) 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 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**Kbytes 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 Avoid Replay Traps — Factor of 3

21264 and 21364 Only

Although programs are expressed using sequential control flow, a program may contain potential, instruction level parallelism (ILP) that can be exploited. For example, evaluation of a complex arithmetic formula may involve multiple data dependencies between variables that allow more than one arithmetic operation to be executed concurrently given sufficient processor resources.

Alpha implementations may use techniques such as register renaming, multiple function units and out-of-order execution to identify and exploit potential ILP. These techniques must respect true data dependencies and preserve architectural program correctness. Register renaming, for example, can be used to preserve correctness of register data dependencies.

The relative time to access data from primary memory is increasing as processor cycle time continues to decrease. Memory latency, the time to access data from memory, has become a dominating factor in program performance. (This is why prefetching and good cache memory behavior are important to good performance.) Alpha implementations may execute memory operations out-of-order as well as register-to-register operations to overlap memory operations with computational operations thereby mitigating the effect of relatively long memory access latencies. Preservation of correctness, however, becomes a more challenging problem to hardware designers.

Consider the following code sequence.

```
addq  R1,R2,R5
stq   R5,0(R7)
ldq   R6,0(R8)
subq  R6,R2,R3
```

An Alpha implementation may choose to speculate the load operation and begin its execution before the add. This dynamic reordering lets the load operation begin early so that its result may be readily available for the subtract instruction. If the load and store access different addresses and the memory data items do not overlap, the load can be successfully and correctly speculated. If the value in R7 equals the value in R8, however, the load and store refer to

the same memory quadword, and speculated execution of the load before the store will produce the wrong value. In order to preserve correctness, an Alpha implementation must be able to detect cases of incorrect speculation and recover. Significant penalties may be associated with such recovery operations.

Compilers should perform pointer analysis to determine if memory operations refer to the same or overlapping data items in memory and to schedule instructions accordingly. Compilers should also schedule code requiring store followed by read sequences such as register-to-stack spill and fill operations. Compiler writers and programmers alike should be mindful of "hidden," non-explicit data dependencies due to pointer aliasing and strive to make memory data dependencies as explicit as possible.

See also *The Compiler Writer's Guide for the Alpha 21264/21364* for more information about replay traps.

A.3.6 Prefetching — Factor of 3

Prefetching is very important (by a factor of 2) for loops dominated by memory latency or bandwidth. The 21264 and 21364 both support three styles of prefetch, but the 21364 has more highly developed support for marking a cache block as having a short temporal cache life with the *evict next* qualifier:

Table A–1 Prefetch Instruction Support

Prefetch Type	Instruction	Processor Support	Description
Normal prefetch	PREFETCH	21264 and 21364	Prefetch for loading data that is expected to be read only. Reduces the latency to read memory.
Normal prefetch, evict next	PREFETCH_EN	21264 and 21364	Normal prefetch and mark for preferential eviction in future cache fills.
Prefetch with modify intent	PREFETCH_M	21264 and 21364	Prefetch for data that will probably be written. Reduces the latency to read memory and bus traffic.
Prefetch with modify intent, evict next	PREFETCH_MEN	21364 only	Prefetch with modify intent and mark for preferential eviction in future cache fills.
Write hint – 64 bytes	WH64	21264 and 21364	Execute if the program intends to write an entire aligned block of 64 bytes. Reduces the amount of memory bandwidth required to write a block of data.
Write hint – 64 bytes, evict next	WH64EN	21364 only	Hint to the processor that the corresponding block should be marked for preferential eviction in future cache fills.

The actual cache eviction policy is implementation-dependent and described in the corresponding implementation's hardware reference manual, available at:

`ftp.compaq.com/pub/products/alphaCPUdocs`

The prefetch instructions and write hints are recognized as prefetches or ignored on implementations that do not support them, so it is always safe for a compiler to use them. The load prefetches have no architecturally visible effect, so inserting prefetches never causes a program error. Because of its more powerful memory system, prefetches on a 21264/21364 have more potential benefit than previous Alpha implementations and unnecessary prefetching is less costly.

Always prefetch ahead at least two cache blocks. Prefetch farther ahead if possible, unless doing so requires more than eight offchip references to be in progress at the same time. That is, for a loop that references n streams, prefetch ahead two blocks or $8/n$ blocks, whichever is greater. Note, however, that for short trip count loops, it may be beneficial to reduce the prefetch distance, so that the prefetched data is likely to be used.

Prefetches to invalid addresses are dismissed by PALcode, so it is safe to prefetch off the end of an array, but it does incur a small (less than 30 cycle) performance penalty. Prefetches can have alignment traps, so align the pointer used to prefetch.

The WH64 instruction sets an aligned 64-byte block to an unknown state. Use WH64 when the program intends to completely write an aligned 64-byte area of memory. Unlike load prefetches, the WH64 instruction modifies data, and it is not safe to execute WH64 off the end of an array. Although a conditional branch can guard the WH64 instruction so that it does not go beyond the end of an array, a better solution is to create a dummy aligned block of 64 bytes of memory on the stack (bitbucket) and use a CMOV instruction to select the bitbucket address when nearing the end of the array. For example:

```
CMPLT  R0,R1,R2 # test if there are at least 64 bytes left
CMOVEQ R2,R3,R4 # if not, overwrite r4 with address of bit bucket
WH64   R4
```

Sections 4.11.8 and 4.11.11 describe the various PREFETCH and WH64 instructions, respectively.

A.4 Code Sequences

The following section describes code sequences.

A.4.1 Aligned Byte/Word (Within Register) Memory Accesses

The instruction sequences given in Chapter 4 for byte-within-register accesses are worst-case code. More importantly, they do not reflect the instructions available with the BWX extension, described in Section 4.6 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 R_x , let $D.lw$ be D rounded down to a multiple of 4 ($(D \text{ div } 4) * 4$), and let $D.mod$ be $D \text{ mod } 4$.

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:

```
LDL    R1,D.lw(Rx)           !
SLL    R1,#56-8*D.mod,R1     !
SRA    R1,#56,R1            !
```

In the common case, the intended sequence for storing an aligned word R_5 is:

```
LDL    R1,D.lw(Rx)           !
INSWL  R5,#D.mod,R3         !
MSKWL  R1,#D.mod,R1         !
BIS    R3,R1,R1             !
STL    R1,D.lw(Rx)           !
```

In the common case, the intended sequence for storing a byte R_5 is:

```
LDL    R1,D.lw(Rx)           !
INSBL  R5,#D.mod,R3         !
MSKBL  R1,#D.mod,R1         !
BIS    R3,R1,R1             !
STL    R1,D.lw(Rx)           !
```

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, multiplication by the reciprocal can closely approximate other divisions by constants.

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. A subroutine can do general quadword division by true variables. 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:

```

LDG   F0,X                ; X = ABCD EFGH
                        ; 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 improves the readability of compiler output and increases the likelihood that an implementation will speed up the stylized form.

A.4.4.1 NOP

The universal NOP form is:

```
UNOP      ==      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:

```
NOP          ==      BIS      R31 ,R31 ,R31
FNOP        ==      CPYS     F31 ,F31 ,F31
```

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:

```
CLR          ==      BIS      R31 ,R31 ,Rx
FCLR        ==      CPYS     F31 ,F31 ,Fx
```

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:

```
MOV #lit8,Ry ==      BIS R31, lit8, 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:

```
val = <sign-extended, 32-bit value>

low = val <15:0>
tmp1 = val - SEXT(low)          ! Account for LDA instruction

high = tmp1 <31:16>
tmp2 = tmp1 - SHIFT_LEFT( SEXT(high,16) )

if tmp2 NE 0 then
    ! original val was in range 7FFF800016..7FFFFFFF16
    extra = 400016
    tmp1 = tmp1 - 4000000016
    high = tmp1 <31:16>
else
    extra = 0
endif
```

The general sequence is:

```
LDA Rdst, low(R31)
LDAH Rdst, extra(Rdst)      ! Omit if extra=0
LDAH Rdst, high(Rdst)      ! Omit if high=0
```

A.4.4.4 Register-to-Register Move

The standard register move forms are:

```
MOV  RX,RY    ==    BIS   RX,RX,RY
FMOV FX,FY    ==    CPYS  FX,FX,FY
```

These move forms 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:

```
NEGz Rx,Ry    ==    SUBz   R31,Rx,Ry    ! z = L or Q
NEGz Fx,Fy    ==    SUBz   F31,Fx,Fy    ! z = F G S or T
FNEGz Fx,Fy   ==    CPYSN  Fx,Fx,Fy    ! z = F G S 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:

```
NOT Rx,Ry     ==    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:

```
OR  Rx,Ry,Rz  ==    BIS   Rx,Ry,Rz
```

The standard alternative to BIC is:

```
ANDNOT Rx,Ry,Rz ==    BIC   Rx,Ry,Rz
```

The standard alternative to EQV is:

```
XORNOT Rx,Ry,Rz ==    EQV   Rx,Ry,Rz
```

A.4.5 Exception and Trap Barriers

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.

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).

EXCB is thus a superset of TRAPB.

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–2, 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 is listed in the right column.

Some instruction mnemonics have synonyms. These differ 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. Examples of synonyms are: BIC/ANDNOT, BIS/OR, and EQV/XORNOT.

Table A–2 Decodable Pseudo-Operations (Stylized Code Forms)

Pseudo-Operation in Listing		Meaning	Actual Instruction Encoding	
BR	target	Branch to target (21-bit signed displacement)	BR	R31, target
CLR	Rx	Clear integer register	BIS	R31, R31, Rx
FABS	Fx, Fy	No-exception generic floating absolute value	CPYS	F31, Fx, Fy
FCLR	Fx	Clear a floating-point register	CPYS	F31, F31, Fx
FMOV	Fx, Fy	Floating-point move	CPYS	Fx, Fx, Fy
FNEG	Fx, Fy	No-exception generic floating negation	CPYSN	Fx, Fx, Fy
FNOP		Floating-point no-op	CPYS	F31, F31, F31
MOV	Lit, Rx	Move 16-bit sign-extended literal to Rx	LDA	Rx, lit(R31)
MOV	{Rx/Lit8}, Ry	Move Rx/8-bit zero-extended literal to Ry	BIS	R31, {Rx/Lit8}, Ry
MF_FPCR	Fx	Move from FPCR	MF_FPCR	Fx, Fx, Fx
MT_FPCR	Fx	Move to FPCR	MT_FPCR	Fx, Fx, Fx
NEGF	Fx, Fy	Negate F_floating	SUBF	F31, Fx, Fy
NEGF/S	Fx, Fy	Negate F_floating, semi-precise	SUBF/S	F31, Fx, Fy

Table A–2 Decodable Pseudo-Operations (Stylized Code Forms) (Continued)

Pseudo-Operation in Listing		Meaning	Actual Instruction Encoding	
NEGG	Fx, Fy	Negate G_floating	SUBG	F31, Fx, Fy
NEGG/S	Fx, Fy	Negate G_floating, semi-precise	SUBG/S	F31, Fx, Fy
NEGL	{Rx/Lit8}, Ry	Negate longword	SUBL	R31, {Rx/Lit}, Ry
NEGL/V	{Rx/Lit8}, Ry	Negate longword with overflow detection	SUBL/V	R31, {Rx/Lit}, Ry
NEGQ	{Rx/Lit8}, Ry	Negate quadword	SUBQ	R31, {Rx/Lit}, Ry
NEGQ/V	{Rx/Lit8}, Ry	Negate quadword with overflow detection	SUBQ/V	R31, {Rx/Lit}, Ry
NEGS	Fx, Fy	Negate S_floating	SUBS	F31, Fx, Fy
NEGS/SU	Fx, Fy	Negate S_floating, software with underflow detection	SUBS/SU	F31, Fx, Fy
NEGS/SUI	Fx, Fy	Negate S_floating, software with underflow and inexact result detection	SUBS/SUI	F31, Fx, Fy
NEGT	Fx, Fy	Negate T_floating	SUBT	F31, Fx, Fy
NEGT/SU	Fx, Fy	Negate T_floating, software with underflow detection	SUBT/SU	F31, Fx, Fy
NEGT/SUI		Negate T_floating, software with underflow and inexact result detection	SUBT/SUI	F31, Fx, Fy
NOP		Integer no-op	BIS	R31, R31, R31
NOT	{Rx/Lit8}, Ry	Logical NOT of Rx/8-bit zero-extended literal storing results in Ry	ORNOT	R31, {Rx/Lit}, Ry
SEXTL	{Rx/Lit8}, Ry	Longword sign-extension of Rx storing results in Ry	ADDL	R31, {Rx/Lit}, Ry
UNOP		Universal NOP for both integer and floating-point code	LDQ_U	R31, 0(Rx)

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:

- At most 40 operate or conditional-branch (not taken) instructions executed in the sequence between LDx_L and STx_C.
- At most two I-stream TB-miss faults. Sequential instruction execution guarantees this.
- No other exceptions triggered during the last execution of the sequence.

Implementation Note:

On all expected implementations, this allows for about 50 μ sec of execution time, even with 100 percent cache misses. This should satisfy any requirement for a 1-msec timer interrupt rate.

IEEE Floating-Point Conformance

A subset of IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 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, compare, and square root. Software routines support 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.

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.

Alpha hardware supports infinity arithmetic with the compare instructions (CMPTyy). When a /S qualifier is included, Alpha hardware may optionally support infinity arithmetic when infinity operands are encountered and, together with overflow disable (OVFD) and division by zero disable (DZED), when infinity is to be generated from finite operands. Otherwise, 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. An OS completion handler (interposed between the hardware and the IEEE user) provides correct infinity arithmetic.

When a /S qualifier is included, Alpha hardware may optionally support NaNs and invalid operations, controlled by the INVD option. Otherwise, Alpha hardware supports NaNs and invalid operations by trapping when a NaN operand is encountered and when a NaN is to be created. An OS completion 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 underflow. When a /S qualifier is included, Alpha hardware may optionally support underflow and denormalized numbers, controlled by the UNFD option. Otherwise, Alpha hardware supports underflow and denormalized numbers by trapping when a denormalized operand is encountered, when a denormalized result is created, and when an underflow occurs. An OS completion handler (interposed between the hardware and the IEEE user) provides correct denormalized and underflow arithmetic.

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. An OS completion 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 signal handlers are supported by compilers and an OS completion handler (interposed between the hardware and the IEEE user), as described in the next section.

B.2 Alpha Support for OS Completion Handlers

Alpha floating-point trap behavior is statically controlled by the /S, /U, and /I mode qualifiers on floating-point instructions. Changing these options usually requires recompiling. Instructions with any valid qualifier combination that includes the /S qualifier can be dynamically controlled by the optional trap disable bits and denormal control bits in the FPCR.

Each Alpha implementation may choose how to distribute support for the completion modes (/S, /SU, /SV, /SUI, and /SVI), between hardware and software. An implementation may minimize hardware complexity by trapping to implementation software for support of exceptions

and non-finites. An implementation may choose increased floating-point performance at the cost of increased hardware complexity by providing hardware support for exceptions and non-finites.

However completion mode support is distributed, application software on any system that meets the Alpha architecture specification will see consistent floating-point semantics because Alpha implementation software provides support for any floating-point feature that is not directly supported by the hardware.

Each Alpha operating system must include an OS completion handler that does software completion of instructions that have any valid qualifier combination that includes the /S qualifier, and that finishes the computation of any floating-point operation that is not completed by the hardware. The OS completion handler is responsible for providing the result specified by the architecture. The handler either continues execution of the application program or signals an exception to the application.

If the exception summary parameter of an arithmetic trap indicates that an instruction requiring software completion caused the trap, the operating system must finish the operation. An OS completion handler uses the register write mask parameter to ignore instructions in the trap shadow and to locate the trigger instruction of the arithmetic trap. The handler then uses the trigger instruction input register values to compute the result in the output register and to record any appropriate signal status. The handler then continues execution with the instruction following the trigger instruction, unless the application has requested execution of an optional signal handler.

It is recommended that the OS completion handler report an enabled IEEE exception to the user application as a fault, rather than as a trap. When reported as a fault, the reported PC points to the trigger instruction, rather than after the trigger instruction. Regardless of whether an enabled fault occurs, it is recommended that the completion trap handler set the result register and status flags to the IEEE standard nontrapping results, as defined in the IEEE Standard in Section 4.7.10. That behavior makes it possible for the user application to continue from a fault by stepping over the trigger instruction.

The Floating-Point Control Register (FPCR) contains several trap disable bits and denormal control bits. Implementation of these bits in the FPCR is optional. A system that includes these bits may choose to complete computations involving non-finite values without the assistance of software completion. Operating systems use these FPCR bits to enable hardware completion of instructions with any valid qualifier combination that includes /S in those cases where the operating system does not require a trap to do exception signaling.

To get the optional full IEEE user trap handler behavior, an OS completion handler must be provided that implements the exception status 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. OS completion handlers can use the FP_Control quadword, along with the floating-point control register (FPCR), to provide various levels of IEEE-compliant behavior.

OS completion handlers provide two options for special handling of denormal numbers in instructions that are compiled with any valid qualifier combination that includes the /S qualifier. These options are controlled by bits defined by implementation software in the IEEE Floating-Point Control (FP_C) Quadword.

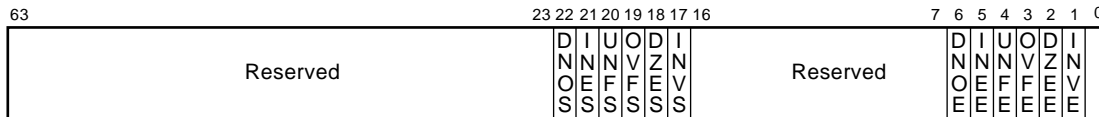
- The first option maps all denormal results to a true zero value. That option is useful for improving the performance of IEEE compliant code that does not need gradual underflow and for mixing IEEE instructions that both include and do not include the /S qualifier.
- A second option treats all denormal input operands as if they were signed zeros. That option is useful for improving the performance of IEEE compliant code that encounters spurious denormal values in uninitialized data.

The optional UNDZ and DNZ (denormal control) bits in the FPCR can assist hardware to improve the performance of these denormal handling options.

B.2.1 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



- The operating system software completion mechanism maintains the FP_C. Therefore, the FP_C affects (and is affected by) only those instructions with any valid qualifier combination that includes the /S qualifier.
- The FP_C quadword is context switched when the operating system switches the thread context. (The FP_C can be placed in a currently switched data structure.)
- Although the operating system can keep the FP_C in a user mode memory location, user code may not directly access the FP_C.
- Integer overflow (IOV) exceptions are controlled by the INVE enable mask bit (FP_C<1>), as allowed by the IEEE standard. Implementation software is responsible for setting the INVS status bit (FP_C<17>) when a CVTTQ or CVTQL instruction traps into the software completion mechanism for integer overflow .
- At process creation, all trap enable flags in the FP_C are clear. The settings of other FP_C bits, defined in Table B–1 as reserved for implementation software, are defined by operating system software.

At other events such as forks or thread creation, and at asynchronous routine calls such as traps and signals, the operating system controls all assigned FP_C bits and those defined as reserved for implementation software.

Table B–1 Floating-Point Control (FP_C) Quadword Bit Summary

Bit	Description
63–48	Reserved for implementation software.
47–23	Reserved for future architecture definition.
22	Denormal operand status (DNOS) A floating arithmetic or conversion operation used a denormal operand value. This status field is left unchanged if the system is treating denormal operand values as if they were signed zero values. If an operation with a denormal operand causes other exceptions, all appropriate status bits are set.
21	Inexact result status (INES) A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.
20	Underflow status (UNFS) A floating arithmetic or conversion operation underflowed the destination exponent.
19	Overflow status (OVFS) A floating arithmetic or conversion operation overflowed the destination exponent.
18	Division by zero status (DZES) An attempt was made to perform a floating divide operation with a divisor of zero.
17	Invalid operation status (INVS) An attempt was made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values were illegal.
16–12	Reserved for implementation software.
11–7	Reserved for future architecture definition.
6	Denormal operand exception enable (DNOE) Initiate an INV exception if a floating arithmetic or conversion operation involves a denormal operand value. This exception does not signal if the system is treating denormal operand values as if they were signed zero values. If an operation can initiate more than one enabled exception, the denormal operand exception has priority.
5	Inexact result enable (INEE) Initiate an INE exception if the result of a floating arithmetic or conversion operation differs from the mathematically exact result.
4	Underflow enable (UNFE) Initiate a UNF exception if a floating arithmetic or conversion operation underflows the destination exponent.

Table B–1 Floating-Point Control (FP_C) Quadword Bit Summary (Continued)

Bit	Description
3	Overflow enable (OVFE) Initiate an OVF exception if a floating arithmetic or conversion operation overflows the destination exponent.
2	Division by zero enable (DZEE) Initiate a DZE exception if an attempt is made to perform a floating divide operation with a divisor of zero.
1	Invalid operation enable (INVE) 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.
0	Reserved for implementation software.

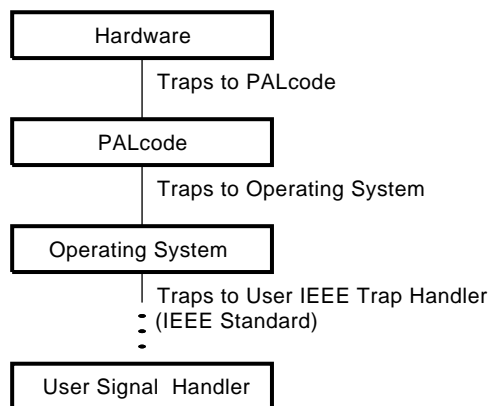
B.3 Mapping to IEEE Standard

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.

The 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, PALcode, the operating-system completion handler, and the user signal 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 signal 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 are: $-\text{MAX}..-\text{MIN}$, -0 , 0 , $+\text{MIN}..+\text{MAX}$

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.

If the DNZ (denormal operands to zero) bit in the FPCR is set or if the OS completion handler is treating denormal operands as zero, then IEEE trap handling is done as if each denormal operand had the corresponding signed zero value.

Table B-2 specifies, for the IEEE /S qualifier modes, which layer does each piece of trap handling. The table describes where the hardware and PALcode can trap to the OS completion handler. However, for IEEE operations with any valid qualifier combination that includes the /S qualifier, the system may choose not to trap to the OS completion handler, provided that any applicable exception is disabled by the trap disable bits in the FPCR and the hardware and PALcode can produce the expected IEEE result as modified by the denormal control bits in the FPCR. See Section 4.7 for more detail on the hardware instruction descriptions.

Table B-2 IEEE Floating-Point Trap Handling

Alpha Instructions	Hardware ¹	PAL-Code	OS Completion Handler	User Signal Handler
FBEQ FBNE FBLT FBLE FBGT FBGE	Bits Only – No Exceptions			
LDS LDT	Bits Only—No Exceptions			
STS STT	Bits Only—No Exceptions			
CPYS CPYSN	Bits Only—No Exceptions			
FCMOVx	Bits Only—No Exceptions			
ADDx SUBx INPUT Exceptions:				
Denormal operand	Trap	Trap	Supply sum	[Denormal Op ²]
$+/-$ Inf operand	Trap	Trap	Supply sum	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
$+Inf + -Inf$	Trap	Trap	Supply QNaN	[Invalid Op]
ADDx SUBx OUTPUT Exceptions:				
Exponent overflow	Trap	Trap	Supply $+/-$ Inf $+/-$ MAX	[Overflow ³] Scale by bias adjust
Exponent underflow and disabled	Supply +0	–	–	– ₄
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply $+/-$ MIN denorm $+/-$ 0	[Underflow ³] Scale by bias adjust

Table B–2 IEEE Floating-Point Trap Handling (Continued)

Alpha Instructions	Hardware ¹	PAL-Code	OS Completion Handler	User Signal Handler
ADDx SUBx OUTPUT Exceptions, Continued:				
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply sum and trap	Trap	–	[Inexact]
MULx INPUT Exceptions:				
Denormal operand	Trap	Trap	Supply prod.	[Denormal Op ²]
+/-Inf operand	Trap	Trap	Supply prod.	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
0 * Inf	Trap	Trap	Supply QNaN	[Invalid Op]
MULx OUTPUT Exceptions:				
Exponent overflow	Trap	Trap	Supply +/-Inf +/-MAX	[Overflow ³] Scale by bias adjust
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and Trap	Trap	Supply +/-MIN denorm +/-0	[Underflow ³] Scale by bias adjust
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply prod. and trap	Trap	–	[Inexact]
DIVx INPUT Exceptions:				
Denormal operand	Trap	Trap	Supply quot.	[Denormal Op ²]
+/-Inf operand	Trap	Trap	Supply quot.	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
0/0 or Inf/Inf	Trap	Trap	Supply QNaN	[Invalid Op]
A/0	Trap	Trap	Supply +/- Inf	[Div. Zero]
DIVx OUTPUT Exceptions:				
Exponent overflow	Trap	Trap	Supply +/-Inf +/- MAX	[Overflow ³] Scale by bias adjust
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/- MIN denorm +/-0	[Underflow ³] Scale by bias adjust
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply quot. and trap	Trap	–	[Inexact]

Table B–2 IEEE Floating-Point Trap Handling (Continued)

Alpha Instructions	Hardware ¹	PAL-Code	OS Completion Handler	User Signal Handler
CMPTEQ CMPTUN INPUT Exceptions:				
Denormal operand	Trap	Trap	Supply (=)	[Denormal Op ²]
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 <	[Denormal Op ²]
QNaN operand	Trap	Trap	Supply False	[Invalid Op]
SNaN operand	Trap	Trap	Supply False	[Invalid Op]
CVTfi INPUT Exceptions:				
Denormal operand	Trap	Trap	Supply Cvt	[Denormal Op ²]
+/-Inf operand	Trap	Trap	Supply 0	[Invalid Op]
QNaN operand	Trap	Trap	Supply 0	–
SNaN operand	Trap	Trap	Supply 0	[Invalid Op]
CVTfi OUTPUT Exceptions:				
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply Cvt and trap	Trap	–	[Inexact]
Integer overflow	Supply Trunc. result and trap if enabled	Trap	–	[Invalid Op ⁵]
CVTif OUTPUT Exceptions:				
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply Cvt and trap	Trap	–	[Inexact]
CVTff INPUT Exceptions:				
Denormal operand	Trap	Trap	Supply Cvt	[Denormal Op ²]
+/-Inf operand	Trap	Trap	Supply Cvt	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
CVTff OUTPUT Exceptions:				
Exponent overflow	Trap	Trap	Supply +/-Inf +/-MAX	[Overflow ³] Scale by bias adjust
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/- MIN denorm +/-0	[Underflow ³] Scale by bias adjust

Table B–2 IEEE Floating-Point Trap Handling (Continued)

Alpha Instructions	Hardware ¹	PAL-Code	OS Completion Handler	User Signal Handler
CVTff OUTPUT Exceptions, continued:				
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply Cvt and trap	Trap	–	[Inexact]
SQRTx INPUT Exceptions:				
Negative nonzero operand	Trap	Trap	Supply QNaN	[Invalid Op]
+/-0	Supply +/-0	–	–	–
+ Denormal operand	Trap	Trap	Supply SQRT	[Denormal Op ²]
– Denormal operand	Trap	Trap	Supply QNaN	[Denormal Op/ Invalid Op]
+ Infinity operand	Trap	Trap	Supply +Inf	–
– Infinity operand	Trap	Trap	Supply QNaN	[Invalid Op]
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
SQRTx OUTPUT Exceptions:				
Exponent overflow	Not possible			
Exponent underflow	Not possible			
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply SQRT	Trap	–	[Inexact]

¹ This column describes the minimum necessary hardware support.

² [Denormal Op] signals have priority over all other signals.

³ [Overflow] and [Underflow] signals have priority over [Inexact] signals.

⁴ An implementation could choose instead to trap to PALcode and have the PALcode supply a zero result on all underflows.

⁵ An implementation could choose instead to trap to PALcode on extreme values and have the PALcode supply a truncated result on all overflows.

Other IEEE operations (software subroutines or sequences of instructions) are listed here for completeness:

- Remainder
- 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. In the charts, the second column is the result when the user signal handler is disabled; the third column is the result when that handler is enabled. The OS completion handler supplies the IEEE default that is specified in the second column. The contents of the Alpha registers contain sufficient information for an enabled user handler to compute the value in the third column.

Table B-3 IEEE Standard Charts

Exception	User Signal Handler Disabled (IEEE Default)	User Signal Handler Enabled (Optional)
Invalid Operation		
(1) Input signaling NaN	Quiet NaN	
(2) Mag. subtract Inf.	Quiet NaN	
(3) 0 * Inf.	Quiet NaN	
(4) 0/0 or Inf/Inf	Quiet NaN	
(5) x REM 0 or Inf REM y	Quiet NaN	
(6) SQRT(negative non-zero)	Quiet NaN	
(7) Cvt to int(ovfl)	Low-order bits	
(8) Cvt to int(Inf, NaN)	0	
(9) Compare unordered	Quiet NaN	
Division by Zero		
x/0, x finite <>0	+/-Inf	
Overflow		
Round nearest	+/-Inf.	Res/2**192 or 1536
Round to zero	+/-MAX	Res/2**192 or 1536
Round to -Inf	+MAX/-Inf	Res/2**192 or 1536
Round to +Inf	+Inf/-MAX	Res/2**192 or 1536
Underflow		
Underflow	0/denorm	Res*2**192 or 1536
Inexact		
Inexact	Rounded	Res

Appendix C

Instruction Summary

This appendix summarizes all instructions and opcodes in the Alpha architecture. All values are in hexadecimal radix.

C.1 Common Architecture Instruction Summary

This section summarizes 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

Instruction Format	Format Symbol	Opcode Notation	Meaning
Branch	Bra	oo	oo is the 6-bit opcode field.
Floating- point	F-P	oo.fff	oo is the 6-bit opcode field. fff is the 11-bit function code field.
Memory	Mem	oo	oo is the 6-bit opcode field.
Memory/ func code	Mfc	oo.ffff	oo is the 6-bit opcode field. ffff is the 16-bit function code in the displacement field.
Memory/ branch	Mbr	oo.h	oo is the 6-bit opcode field. h is the high-order two bits of the displacement field.
Operate	Opr	oo.ff	oo is the 6-bit opcode field. ff is the 7-bit function code field.
PALcode	Pcd	oo	oo is the 6-bit opcode field; the particular PALcode instruction is specified in the 26-bit function code field.

Table C–2 shows qualifiers for operate format instructions. Qualifiers for IEEE and VAX floating-point instructions are shown in Sections C.2 and C.3, respectively.

Table C–2 Common Architecture Instructions

Mnemonic	Format	Opcode	Description
ADDF	F-P	15.080	Add F_floating
ADDG	F-P	15.0A0	Add G_floating
ADDL	Opr	10.00	Add longword
ADDL/V		10.40	
ADDQ	Opr	10.20	Add quadword
ADDQ/V		10.60	
ADDS	F-P	16.080	Add S_floating
ADDT	F-P	16.0A0	Add T_floating
AMASK	Opr	11.61	Architecture mask
AND	Opr	11.00	Logical product
BEQ	Bra	39	Branch if = zero
BGE	Bra	3E	Branch if \geq zero
BGT	Bra	3F	Branch if > zero
BIC	Opr	11.08	Bit clear
BIS	Opr	11.20	Logical sum
BLBC	Bra	38	Branch if low bit clear
BLBS	Bra	3C	Branch if low bit set
BLE	Bra	3B	Branch if \leq zero
BLT	Bra	3A	Branch if < zero
BNE	Bra	3D	Branch if \neq zero
BR	Bra	30	Unconditional branch
BSR	Mbr	34	Branch to subroutine
CALL_PAL	Pcd	00	Trap to PALcode
CMOVEQ	Opr	11.24	CMOVE if = zero
CMOVGE	Opr	11.46	CMOVE if \geq zero
CMOVGT	Opr	11.66	CMOVE if > zero
CMOVLBC	Opr	11.16	CMOVE if low bit clear
CMOVLBS	Opr	11.14	CMOVE if low bit set
CMOVLE	Opr	11.64	CMOVE if \leq zero
CMOVLT	Opr	11.44	CMOVE if < zero
CMOVNE	Opr	11.26	CMOVE if \neq zero
CMPBGE	Opr	10.0F	Compare byte
CMPEQ	Opr	10.2D	Compare signed quadword equal
CMPGEQ	F-P	15.0A5	Compare G_floating equal
CMPGLE	F-P	15.0A7	Compare G_floating less than or equal
CMPGLT	F-P	15.0A6	Compare G_floating less than
CMPLE	Opr	10.6D	Compare signed quadword less than or equal
CMPLT	Opr	10.4D	Compare signed quadword less than

Table C–2 Common Architecture Instructions (Continued)

Mnemonic	Format	Opcode	Description
CMPTEQ	F-P	16.0A5	Compare T_floating equal
CMPTLE	F-P	16.0A7	Compare T_floating less than or equal
CMPTLT	F-P	16.0A6	Compare T_floating less than
CMPTUN	F-P	16.0A4	Compare T_floating unordered
CMPULE	Opr	10.3D	Compare unsigned quadword less than or equal
CMPULT	Opr	10.1D	Compare unsigned quadword less than
CPYS	F-P	17.020	Copy sign
CPYSE	F-P	17.022	Copy sign and exponent
CPYSN	F-P	17.021	Copy sign negate
CTLZ	Opr	1C.32	Count leading zero
CTPOP	Opr	1C.30	Count population
CTTZ	Opr	1C.33	Count trailing zero
CVTDG	F-P	15.09E	Convert D_floating to G_floating
CVTGD	F-P	15.0AD	Convert G_floating to D_floating
CVTGF	F-P	15.0AC	Convert G_floating to F_floating
CVTGQ	F-P	15.0AF	Convert G_floating to quadword
CVTLQ	F-P	17.010	Convert longword to quadword
CVTQF	F-P	15.0BC	Convert quadword to F_floating
CVTQG	F-P	15.0BE	Convert quadword to G_floating
CVTQL	F-P	17.030	Convert quadword to longword
CVTQS	F-P	16.0BC	Convert quadword to S_floating
CVTQT	F-P	16.0BE	Convert quadword to T_floating
CVTST	F-P	16.2AC	Convert S_floating to T_floating
CVTTQ	F-P	16.0AF	Convert T_floating to quadword
CVTTS	F-P	16.0AC	Convert T_floating to S_floating
DIVF	F-P	15.083	Divide F_floating
DIVG	F-P	15.0A3	Divide G_floating
DIVS	F-P	16.083	Divide S_floating
DIVT	F-P	16.0A3	Divide T_floating
ECB	Mfc	18.E800	Evict cache block
EQV	Opr	11.48	Logical equivalence
EXCB	Mfc	18.0400	Exception barrier
EXTBL	Opr	12.06	Extract byte low
EXTLH	Opr	12.6A	Extract longword high
EXTLL	Opr	12.26	Extract longword low
EXTQH	Opr	12.7A	Extract quadword high
EXTQL	Opr	12.36	Extract quadword low
EXTWH	Opr	12.5A	Extract word high
EXTWL	Opr	12.16	Extract word low
FBEQ	Bra	31	Floating branch if = zero

Table C–2 Common Architecture Instructions (Continued)

Mnemonic	Format	Opcode	Description
FBGE	Bra	36	Floating branch if \geq zero
FBGT	Bra	37	Floating branch if $>$ zero
FBLE	Bra	33	Floating branch if \leq zero
FBLT	Bra	32	Floating branch if $<$ zero
FBNE	Bra	35	Floating branch if \neq zero
FCMOVEQ	F-P	17.02A	FCMOVE if = zero
FCMOVGE	F-P	17.02D	FCMOVE if \geq zero
FCMOVGT	F-P	17.02F	FCMOVE if $>$ zero
FCMOVLE	F-P	17.02E	FCMOVE if \leq zero
FCMOVLT	F-P	17.02C	FCMOVE if $<$ zero
FCMOVNE	F-P	17.02B	FCMOVE if \neq zero
FETCH	Mfc	18.8000	Prefetch data
FETCH_M	Mfc	18.A000	Prefetch data, modify intent
FTOIS	F-P	1C.78	Floating to integer move, S_floating
FTOIT	F-P	1C.70	Floating to integer move, T_floating
IMPLVER	Opr	11.6C	Implementation version
INSBL	Opr	12.0B	Insert byte low
INSLH	Opr	12.67	Insert longword high
INSLL	Opr	12.2B	Insert longword low
INSQH	Opr	12.77	Insert quadword high
INSQL	Opr	12.3B	Insert quadword low
INSWH	Opr	12.57	Insert word high
INSWL	Opr	12.1B	Insert word low
ITOFF	F-P	14.014	Integer to floating move, F_floating
ITOFS	F-P	14.004	Integer to floating move, S_floating
ITOFT	F-P	14.024	Integer to floating move, T_floating
JMP	Mbr	1A.0	Jump
JSR	Mbr	1A.1	Jump to subroutine
JSR_COROUTINE	Mbr	1A.3	Jump to subroutine return
LDA	Mem	08	Load address
LDAH	Mem	09	Load address high
LDBU	Mem	0A	Load zero-extended byte
LDWU	Mem	0C	Load zero-extended word
LDF	Mem	20	Load F_floating
LDG	Mem	21	Load G_floating
LDL	Mem	28	Load sign-extended longword
LDL_L	Mem	2A	Load sign-extended longword locked
LDQ	Mem	29	Load quadword
LDQ_L	Mem	2B	Load quadword locked
LDQ_U	Mem	0B	Load unaligned quadword

Table C–2 Common Architecture Instructions (Continued)

Mnemonic	Format	Opcode	Description
LDS	Mem	22	Load S_floating
LDT	Mem	23	Load T_floating
MAXSB8	Opr	1C.3E	Vector signed byte maximum
MAXSW4	Opr	1C.3F	Vector signed word maximum
MAXUB8	Opr	1C.3C	Vector unsigned byte maximum
MAXUW4	Opr	1C.3D	Vector unsigned word maximum
MB	Mfc	18.4000	Memory barrier
MF_FPCR	F-P	17.025	Move from FPCR
MINSB8	Opr	1C.38	Vector signed byte minimum
MINSW4	Opr	1C.39	Vector signed word minimum
MINUB8	Opr	1C.3A	Vector unsigned byte minimum
MINUW4	Opr	1C.3B	Vector unsigned word minimum
MSKBL	Opr	12.02	Mask byte low
MSKLBH	Opr	12.62	Mask longword high
MSKLL	Opr	12.22	Mask longword low
MSKQH	Opr	12.72	Mask quadword high
MSKQL	Opr	12.32	Mask quadword low
MSKWH	Opr	12.52	Mask word high
MSKWL	Opr	12.12	Mask word low
MT_FPCR	F-P	17.024	Move to FPCR
MULF	F-P	15.082	Multiply F_floating
MULG	F-P	15.0A2	Multiply G_floating
MULL	Opr	13.00	Multiply longword
MULL/V		13.40	
MULQ	Opr	13.20	Multiply quadword
MULQ/V		13.60	
MULS	F-P	16.082	Multiply S_floating
MULT	F-P	16.0A2	Multiply T_floating
ORNOT	Opr	11.28	Logical sum with complement
PERR	Opr	1C.31	Pixel error
PKLB	Opr	1C.37	Pack longwords to bytes
PKWB	Opr	1C.36	Pack words to bytes
PREFETCH	Mem	28 ¹	Prefetch a cache block
PREFETCH_EN	Mem	29 ¹	Prefetch a cache block, evict next
PREFETCH_M	Mem	22 ¹	Prefetch a cache block, modify intent
PREFETCH_MEN	Mem	23 ¹	Prefetch cache block, modify intent, evict next
RC	Mfc	18.E000	Read and clear
RET	Mbr	1A.2	Return from subroutine
RPCC	Mfc	18.C000	Read process cycle counter
RS	Mfc	18.F000	Read and set

Table C–2 Common Architecture Instructions (Continued)

Mnemonic	Format	Opcode	Description
S4ADDL	Opr	10.02	Scaled add longword by 4
S4ADDQ	Opr	10.22	Scaled add quadword by 4
S4SUBL	Opr	10.0B	Scaled subtract longword by 4
S4SUBQ	Opr	10.2B	Scaled subtract quadword by 4
S8ADDL	Opr	10.12	Scaled add longword by 8
S8ADDQ	Opr	10.32	Scaled add quadword by 8
S8SUBL	Opr	10.1B	Scaled subtract longword by 8
S8SUBQ	Opr	10.3B	Scaled subtract quadword by 8
SEXTB	Opr	1C.00	Sign extend byte
SEXTW	Opr	1C.01	Sign extend word
SLL	Opr	12.39	Shift left logical
SQRTF	F-P	14.08A	Square root F_floating
SQRTG	F-P	14.0AA	Square root G_floating
SQRTS	F-P	14.08B	Square root S_floating
SQRTT	F-P	14.0AB	Square root T_floating
SRA	Opr	12.3C	Shift right arithmetic
SRL	Opr	12.34	Shift right logical
STB	Mem	0E	Store byte
STF	Mem	24	Store F_floating
STG	Mem	25	Store G_floating
STS	Mem	26	Store S_floating
STL	Mem	2C	Store longword
STL_C	Mem	2E	Store longword conditional
STQ	Mem	2D	Store quadword
STQ_C	Mem	2F	Store quadword conditional
STQ_U	Mem	0F	Store unaligned quadword
STT	Mem	27	Store T_floating
STW	Mem	0D	Store word
SUBF	F-P	15.081	Subtract F_floating
SUBG	F-P	15.0A1	Subtract G_floating
SUBL	Opr	10.09	Subtract longword
SUBL/V		10.49	
SUBQ	Opr	10.29	Subtract quadword
SUBQ/V		10.69	
SUBS	F-P	16.081	Subtract S_floating
SUBT	F-P	16.0A1	Subtract T_floating
TRAPB	Mfc	18.0000	Trap barrier
UMULH	Opr	13.30	Unsigned multiply quadword high
UNPKBL	Opr	1C.35	Unpack bytes to longwords
UNPKBW	Opr	1C.34	Unpack bytes to words

Table C–2 Common Architecture Instructions (Continued)

Mnemonic	Format	Opcode	Description
WH64	Mfc	18.F800	Write hint — 64 bytes
WH64EN	Mfc	18.FC00	Write hint — 64 bytes, evict next
WMB	Mfc	18.4400	Write memory barrier
XOR	Opr	11.40	Logical difference
ZAP	Opr	12.30	Zero bytes
ZAPNOT	Opr	12.31	Zero bytes not

¹ PREFETCHx instructions share opcodes with the corresponding load instructions. The PREFETCHx instructions are distinguished by, in each case, the Fa or Ra operand set to 31.

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 16₁₆, except for SQRTS and SQRTT, which are opcode 14₁₆.

Table C–3 IEEE Floating-Point Instruction Function Codes

	None	/C	/M	/D	/U	/UC	/UM	/UD
ADDS	080	000	040	0C0	180	100	140	1C0
ADDT	0A0	020	060	0E0	1A0	120	160	1E0
CMPTEQ	0A5							
CMPTLT	0A6							
CMPTLE	0A7							
CMPTUN	0A4							
CVTQS	0BC	03C	07C	0FC				
CVTQT	0BE	03E	07E	0FE				
CVTST	See below							
CVTTQ	See below							
CVTTS	0AC	02C	06C	0EC	1AC	12C	16C	1EC
DIVS	083	003	043	0C3	183	103	143	1C3
DIVT	0A3	023	063	0E3	1A3	123	163	1E3
MULS	082	002	042	0C2	182	102	142	1C2
MULT	0A2	022	062	0E2	1A2	122	162	1E2
SQRTS	08B	00B	04B	0CB	18B	10B	14B	1CB
SQRTT	0AB	02B	06B	0EB	1AB	12B	16B	1EB
SUBS	081	001	041	0C1	181	101	141	1C1
SUBT	0A1	021	061	0E1	1A1	121	161	1E1

Table C-3 IEEE Floating-Point Instruction Function Codes (Continued)

	/SU	/SUC	/SUM	/SUD	/SUI	/SUIC	/SUIM	/SUID
ADDS	580	500	540	5C0	780	700	740	7C0
ADDT	5A0	520	560	5E0	7A0	720	760	7E0
CMPTEQ	5A5							
CMPTLT	5A6							
CMPTLE	5A7							
CMPTUN	5A4							
CVTQS					7BC	73C	77C	7FC
CVTQT					7BE	73E	77E	7FE
CVTTS	5AC	52C	56C	5EC	7AC	72C	76C	7EC
DIVS	583	503	543	5C3	783	703	743	7C3
DIVT	5A3	523	563	5E3	7A3	723	763	7E3
MULS	582	502	542	5C2	782	702	742	7C2
MULT	5A2	522	562	5E2	7A2	722	762	7E2
SQRTS	58B	50B	54B	5CB	78B	70B	74B	7CB
SQRTT	5AB	52B	56B	5EB	7AB	72B	76B	7EB
SUBS	581	501	541	5C1	781	701	741	7C1
SUBT	5A1	521	561	5E1	7A1	721	761	7E1

	None	/S
CVTST	2AC	6AC

	None	/C	/V	/VC	/SV	/SVC	/SVI	/SVIC
CVTTQ	0AF	02F	1AF	12F	5AF	52F	7AF	72F

	/D	/VD	/SVD	/SVID	/M	/VM	/SVM	/SVIM
CVTTQ	0EF	1EF	5EF	7EF	06F	16F	56F	76F

Programming Note:

To use CMPTxx with software completion trap handling, specify the /SU IEEE trap mode, even though an underflow trap is not possible. To use CVTQS or CVTQT with software completion trap handling, 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

	None	/C	/U	/UC	/S	/SC	/SU	/SUC
ADDF	080	000	180	100	480	400	580	500
CVTDG	09E	01E	19E	11E	49E	41E	59E	51E
ADDG	0A0	020	1A0	120	4A0	420	5A0	520
CMPGEQ	0A5				4A5			
CMPGLT	0A6				4A6			
CMPGLE	0A7				4A7			
CVTGF	0AC	02C	1AC	12C	4AC	42C	5AC	52C
CVTGD	0AD	02D	1AD	12D	4AD	42D	5AD	52D
CVTGQ	See below							
CVTQF	0BC	03C						
CVTQG	0BE	03E						
DIVF	083	003	183	103	483	403	583	503
DIVG	0A3	023	1A3	123	4A3	423	5A3	523
MULF	082	002	182	102	482	402	582	502
MULG	0A2	022	1A2	122	4A2	422	5A2	522
SQRTF	08A	00A	18A	10A	48A	40A	58A	50A
SQRTG	0AA	02A	1AA	12A	4AA	42A	5AA	52A
SUBF	081	001	181	101	481	401	581	501
SUBG	0A1	021	1A1	121	4A1	421	5A1	521

	None	/C	/V	/VC	/S	/SC	/SV	/SVC
CVTGQ	0AF	02F	1AF	12F	4AF	42F	5AF	52F

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

	None	/V	/SV
CPYS	020		
CPYSE	022		
CPYSN	021		

Table C–5: Independent Floating-Point Instruction Function Codes (Continued)

	None	/V	/SV
CVTLQ	010		
CVTQL	030	130	530
FCMOVEQ	02A		
FCMOVGE	02D		
FCMOVGT	02F		
FCMOVLE	02E		
FCMOVLT	02C		
MF_FPCR	025		
MT_FPCR	024		

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

	00	08	10	18	20	28	30	38
0/8	PAL* (pal)	LDA (mem)	INTA* (op)	MISC* (mem)	LDF (mem)	LDL (mem)	BR (br)	BLBC (br)
1/9	Res	LDAH (mem)	INTL* (op)	\PAL\ (mem)	LDG (mem)	LDQ (mem)	FBEQ (br)	BEQ (br)
2/A	Res	LDBU (mem)	INTS* (op)	JSR* (mem)	LDS (mem)	LDL_L (mem)	FBLT (br)	BLT (br)
3/B	Res	LDQ_U (mem)	INTM* (op)	\PAL\ (mem)	LDT (mem)	LDQ_L (mem)	FBLE (br)	BLE (br)

Table C–6: Opcode Summary (Continued)

	00	08	10	18	20	28	30	38
4/C	Res	LDWU (mem)	ITFP*	FPTI*	STF (mem)	STL (mem)	BSR (br)	BLBS (br)
5/D	Res	STW (mem)	FLTV* (op)	\PAL\ (op)	STG (mem)	STQ (mem)	FBNE (br)	BNE (br)
6/E	Res	STB (mem)	FLTI* (op)	\PAL\ (op)	STS (mem)	STL_C (mem)	FBGE (br)	BGE (br)
7/F	Res	STQ_U (mem)	FLTL* (op)	\PAL\ (op)	STT (mem)	STQ_C (mem)	FBGT (br)	BGT (br)

Table C–7: Key to Opcode Summary

Symbol	Meaning
FLTI*	IEEE floating-point instruction opcodes
FLTL*	Floating-point Operate instruction opcodes
FLTV*	VAX floating-point instruction opcodes
FPTI*	Floating-point to integer register move opcodes
INTA*	Integer arithmetic instruction opcodes
INTL*	Integer logical instruction opcodes
INTM*	Integer multiply instruction opcodes
INTS*	Integer shift instruction opcodes
ITFP*	Integer to floating-point register move opcodes
JSR*	Jump instruction opcodes
MISC*	Miscellaneous instruction opcodes
PAL*	PALcode instruction (CALL_PAL) opcodes
\PAL\ (op)	Reserved for PALcode
Res	Reserved for Compaq

C.6 Common Architecture Opcodes in Numerical Order

Table C–8 lists the common architecture opcodes in numerical order.

Table C–8 Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
00	CALL_PAL	01	OPC01	02	OPC02
03	OPC03	04	OPC04	05	OPC05
06	OPC06	07	OPC07	08	LDA
09	LDAH	0A	LDBU	0B	LDQ_U
0C	LDWU	0D	STW	0E	STB
0F	STQ_U	10.00	ADDL	10.02	S4ADDL
10.09	SUBL	10.0B	S4SUBL	10.0F	CMPBGE
10.12	S8ADDL	10.1B	S8SUBL	10.1D	CMPULT
10.20	ADDQ	10.22	S4ADDQ	10.29	SUBQ
10.2B	S4SUBQ	10.2D	CMPEQ	10.32	S8ADDQ
10.3B	S8SUBQ	10.3D	CMPULE	10.40	ADDL/V
10.49	SUBL/V	10.4D	CMPLT	10.60	ADDQ/V
10.69	SUBQ/V	10.6D	CMPLE	11.00	AND
11.08	BIC	11.14	CMOVLBS	11.16	CMOVLBC
11.20	BIS	11.24	CMOVEQ	11.26	CMOVNE
11.28	ORNOT	11.40	XOR	11.44	CMOVLТ
11.46	CMOVGE	11.48	EQV	11.61	AMASK
11.64	CMOVLE	11.66	CMOVGT	11.6C	IMPLVER
12.02	MSKBL	12.06	EXTBL	12.0B	INSBL
12.12	MSKWL	12.16	EXTWL	12.1B	INSWL
12.22	MSKLL	12.26	EXTLL	12.2B	INSLЛ
12.30	ZAP	12.31	ZAPNOT	12.32	MSKQL
12.34	SRL	12.36	EXTQL	12.39	SLL
12.3B	INSQL	12.3C	SRA	12.52	MSKWH
12.57	INSWH	12.5A	EXTWH	12.62	MSKЛH
12.67	INSLH	12.6A	EXTLH	12.72	MSKQH
12.77	INSQH	12.7A	EXTQH	13.00	MULL
13.20	MULQ	13.30	UMULH	13.40	MULL/V
13.60	MULQ/V	14.004	ITOFs	14.00A	SQRTF/C
14.00B	SQRTS/C	14.014	ITOFF	14.024	ITOFT
14.02A	SQRTG/C	14.02B	SQRTT/C	14.04B	SQRTS/M
14.06B	SQRTT/M	14.08A	SQRTF	14.08B	SQRTS
14.0AA	SQRTG	14.0AB	SQRTT	14.0CB	SQRTS/D
14.0EB	SQRTT/D	14.10A	SQRTF/UC	14.10B	SQRTS/UC
14.12A	SQRTG/UC	14.12B	SQRTT/UC	14.14B	SQRTS/UM
14.16B	SQRTT/UM	14.18A	SQRTF/U	14.18B	SQRTS/U
14.1AA	SQRTG/U	14.1AB	SQRTT/U	14.1CB	SQRTS/UD

Table C–8 Common Architecture Opcodes in Numerical Order (Continued)

Opcode		Opcode		Opcode	
14.1EB	SQRTT/UD	14.40A	SQRTF/SC	14.42A	SQRTG/SC
14.48A	SQRTF/S	14.4AA	SQRTG/S	14.50A	SQRTF/SUC
14.50B	SQRTS/SUC	14.52A	SQRTG/SUC	14.52B	SQRTT/SUC
14.54B	SQRTS/SUM	14.56B	SQRTT/SUM	14.58A	SQRTF/SU
14.58B	SQRTS/SU	14.5AA	SQRTG/SU	14.5AB	SQRTT/SU
14.5CB	SQRTS/SUD	14.5EB	SQRTT/SUD	14.70B	SQRTS/SUIC
14.72B	SQRTT/SUIC	14.74B	SQRTS/SUIM	14.76B	SQRTT/SUIM
14.78B	SQRTS/SUI	14.7AB	SQRTT/SUI	14.7CB	SQRTS/SUID
14.7EB	SQRTT/SUID	15.000	ADDF/C	15.001	SUBF/C
15.002	MULF/C	15.003	DIVF/C	15.01E	CVTDG/C
15.020	ADDG/C	15.021	SUBG/C	15.022	MULG/C
15.023	DIVG/C	15.02C	CVTGF/C	15.02D	CVTGD/C
15.02F	CVTGQ/C	15.03C	CVTQF/C	15.03E	CVTQG/C
15.080	ADDF	15.081	SUBF	15.082	MULF
15.083	DIVF	15.09E	CVTDG	15.0A0	ADDG
15.0A1	SUBG	15.0A2	MULG	15.0A3	DIVG
15.0A5	CMPGEQ	15.0A6	CMPGLT	15.0A7	CMPGLE
15.0AC	CVTGF	15.0AD	CVTGD	15.0AF	CVTGQ
15.0BC	CVTQF	15.0BE	CVTQG	15.100	ADDF/UC
15.101	SUBF/UC	15.102	MULF/UC	15.103	DIVF/UC
15.11E	CVTDG/UC	15.120	ADDG/UC	15.121	SUBG/UC
15.122	MULG/UC	15.123	DIVG/UC	15.12C	CVTGF/UC
15.12D	CVTGD/UC	15.12F	CVTGQ/VC	15.180	ADDF/U
15.181	SUBF/U	15.182	MULF/U	15.183	DIVF/U
15.19E	CVTDG/U	15.1A0	ADDG/U	15.1A1	SUBG/U
15.1A2	MULG/U	15.1A3	DIVG/U	15.1AC	CVTGF/U
15.1AD	CVTGD/U	15.1AF	CVTGQ/V	15.400	ADDF/SC
15.401	SUBF/SC	15.402	MULF/SC	15.403	DIVF/SC
15.41E	CVTDG/SC	15.420	ADDG/SC	15.421	SUBG/SC
15.422	MULG/SC	15.423	DIVG/SC	15.42C	CVTGF/SC
15.42D	CVTGD/SC	15.42F	CVTGQ/SC	15.480	ADDF/S
15.481	SUBF/S	15.482	MULF/S	15.483	DIVF/S
15.49E	CVTDG/S	15.4A0	ADDG/S	15.4A1	SUBG/S
15.4A2	MULG/S	15.4A3	DIVG/S	15.4A5	CMPGEQ/S
15.4A6	CMPGLT/S	15.4A7	CMPGLE/S	15.4AC	CVTGF/S
15.4AD	CVTGD/S	15.4AF	CVTGQ/S	15.500	ADDF/SUC
15.501	SUBF/SUC	15.502	MULF/SUC	15.503	DIVF/SUC
15.51E	CVTDG/SUC	15.520	ADDG/SUC	15.521	SUBG/SUC
15.522	MULG/SUC	15.523	DIVG/SUC	15.52C	CVTGF/SUC
15.52D	CVTGD/SUC	15.52F	CVTGQ/SVC	15.580	ADDF/SU
15.581	SUBF/SU	15.582	MULF/SU	15.583	DIVF/SU

Table C–8 Common Architecture Opcodes in Numerical Order (Continued)

Opcode		Opcode		Opcode	
15.59E	CVTDG/SU	15.5A0	ADDG/SU	15.5A1	SUBG/SU
15.5A2	MULG/SU	15.5A3	DIVG/SU	15.5AC	CVTGF/SU
15.5AD	CVTGD/SU	15.5AF	CVTGQ/SV	16.000	ADDS/C
16.001	SUBS/C	16.002	MULS/C	16.003	DIVS/C
16.020	ADDT/C	16.021	SUBT/C	16.022	MULT/C
16.023	DIVT/C	16.02C	CVTTS/C	16.02F	CVTTQ/C
16.03C	CVTQS/C	16.03E	CVTQT/C	16.040	ADDS/M
16.041	SUBS/M	16.042	MULS/M	16.043	DIVS/M
16.060	ADDT/M	16.061	SUBT/M	16.062	MULT/M
16.063	DIVT/M	16.06C	CVTTS/M	16.06F	CVTTQ/M
16.07C	CVTQS/M	16.07E	CVTQT/M	16.080	ADDS
16.081	SUBS	16.082	MULS	16.083	DIVS
16.0A0	ADDT	16.0A1	SUBT	16.0A2	MULT
16.0A3	DIVT	16.0A4	CMPTUN	16.0A5	CMPTEQ
16.0A6	CMPTLT	16.0A7	CMPTLE	16.0AC	CVTTS
16.0AF	CVTTQ	16.0BC	CVTQS	16.0BE	CVTQT
16.0C0	ADDS/D	16.0C1	SUBS/D	16.0C2	MULS/D
16.0C3	DIVS/D	16.0E0	ADDT/D	16.0E1	SUBT/D
16.0E2	MULT/D	16.0E3	DIVT/D	16.0EC	CVTTS/D
16.0EF	CVTTQ/D	16.0FC	CVTQS/D	16.0FE	CVTQT/D
16.100	ADDS/UC	16.101	SUBS/UC	16.102	MULS/UC
16.103	DIVS/UC	16.120	ADDT/UC	16.121	SUBT/UC
16.122	MULT/UC	16.123	DIVT/UC	16.12C	CVTTS/UC
16.12F	CVTTQ/VC	16.140	ADDS/UM	16.141	SUBS/UM
16.142	MULS/UM	16.143	DIVS/UM	16.160	ADDT/UM
16.161	SUBT/UM	16.162	MULT/UM	16.163	DIVT/UM
16.16C	CVTTS/UM	16.16F	CVTTQ/VM	16.180	ADDS/U
16.181	SUBS/U	16.182	MULS/U	16.183	DIVS/U
16.1A0	ADDT/U	16.1A1	SUBT/U	16.1A2	MULT/U
16.1A3	DIVT/U	16.1AC	CVTTS/U	16.1AF	CVTTQ/V
16.1C0	ADDS/UD	16.1C1	SUBS/UD	16.1C2	MULS/UD
16.1C3	DIVS/UD	16.1E0	ADDT/UD	16.1E1	SUBT/UD
16.1E2	MULT/UD	16.1E3	DIVT/UD	16.1EC	CVTTS/UD
16.1EF	CVTTQ/VD	16.2AC	CVTST	16.500	ADDS/SUC
16.501	SUBS/SUC	16.502	MULS/SUC	16.503	DIVS/SUC
16.520	ADDT/SUC	16.521	SUBT/SUC	16.522	MULT/SUC
16.523	DIVT/SUC	16.52C	CVTTS/SUC	16.52F	CVTTQ/SVC
16.540	ADDS/SUM	16.541	SUBS/SUM	16.542	MULS/SUM
16.543	DIVS/SUM	16.560	ADDT/SUM	16.561	SUBT/SUM
16.562	MULT/SUM	16.563	DIVT/SUM	16.56C	CVTTS/SUM
16.56F	CVTTQ/SVM	16.580	ADDS/SU	16.581	SUBS/SU

Table C–8 Common Architecture Opcodes in Numerical Order (Continued)

Opcode		Opcode		Opcode	
16.582	MULS/SU	16.583	DIVS/SU	16.5A0	ADDT/SU
16.5A1	SUBT/SU	16.5A2	MULT/SU	16.5A3	DIVT/SU
16.5A4	CMPTUN/SU	16.5A5	CMPTEQ/SU	16.5A6	CMPTLT/SU
16.5A7	CMPTLE/SU	16.5AC	CVTTS/SU	16.5AF	CVTTQ/SV
16.5C0	ADDS/SUD	16.5C1	SUBS/SUD	16.5C2	MULS/SUD
16.5C3	DIVS/SUD	16.5E0	ADDT/SUD	16.5E1	SUBT/SUD
16.5E2	MULT/SUD	16.5E3	DIVT/SUD	16.5EC	CVTTS/SUD
16.5EF	CVTTQ/SVD	16.6AC	CVTST/S	16.700	ADDS/SUIC
16.701	SUBS/SUIC	16.702	MULS/SUIC	16.703	DIVS/SUIC
16.720	ADDT/SUIC	16.721	SUBT/SUIC	16.722	MULT/SUIC
16.723	DIVT/SUIC	16.72C	CVTTS/SUIC	16.72F	CVTTQ/SVIC
16.73C	CVTQS/SUIC	16.73E	CVTQT/SUIC	16.740	ADDS/SUIM
16.741	SUBS/SUIM	16.742	MULS/SUIM	16.743	DIVS/SUIM
16.760	ADDT/SUIM	16.761	SUBT/SUIM	16.762	MULT/SUIM
16.763	DIVT/SUIM	16.76C	CVTTS/SUIM	16.76F	CVTTQ/SVIM
16.77C	CVTQS/SUIM	16.77E	CVTQT/SUIM	16.780	ADDS/SUI
16.781	SUBS/SUI	16.782	MULS/SUI	16.783	DIVS/SUI
16.7A0	ADDT/SUI	16.7A1	SUBT/SUI	16.7A2	MULT/SUI
16.7A3	DIVT/SUI	16.7AC	CVTTS/SUI	16.7AF	CVTTQ/SVI
16.7BC	CVTQS/SUI	16.7BE	CVTQT/SUI	16.7C0	ADDS/SUID
16.7C1	SUBS/SUID	16.7C2	MULS/SUID	16.7C3	DIVS/SUID
16.7E0	ADDT/SUID	16.7E1	SUBT/SUID	16.7E2	MULT/SUID
16.7E3	DIVT/SUID	16.7EC	CVTTS/SUID	16.7EF	CVTTQ/SVID
16.7FC	CVTQS/SUID	16.7FE	CVTQT/SUID	17.010	CVTLQ
17.020	CPYS	17.021	CPYSN	17.022	CPYSE
17.024	MT_FPCR	17.025	MF_FPCR	17.02A	FCMOVEQ
17.02B	FCMOVNE	17.02C	FCMOVLT	17.02D	FCMOVGE
17.02E	FCMOVLE	17.02F	FCMOVGT	17.030	CVTQL
17.130	CVTQL/V	17.530	CVTQL/SV	18.0000	TRAPB
18.0400	EXCB	18.4000	MB	18.4400	WMB
18.8000	FETCH	18.A000	FETCH_M	18.C000	RPCC
18.E000	RC	18.E800	ECB	18.F000	RS
18.F800	WH64	18.FC00	WH64EN	19	PAL19
1A.0	JMP	1A.1	JSR	1A.2	RET
1A.3	JSR_COROUTINE	1B	PAL1B	1C.00	SEXTB
1C.01	SEXTW	1C.30	CTPOP	1C.31	PERR
1C.32	CTLZ	1C.33	CTTZ	1C.34	UNPKBW
1C.35	UNPKBL	1C.36	PKWB	1C.37	PKLB
1C.38	MINSB8	1C.39	MINSW4	1C.3A	MINUB8
1C.3B	MINUW4	1C.3C	MAXUB8	1C.3D	MAXUW4
1C.3E	MAXSB8	1C.3F	MAXSW4	1C.70	FTOIT

Table C–8 Common Architecture Opcodes in Numerical Order (Continued)

Opcode		Opcode		Opcode	
1C.78	FTOIS	1D	PAL1D	1E	PAL1E
1F	PAL1F	20	LDF	21	LDG
22	LDS	22 ¹	PREFETCH_M	23	LDT
23 ¹	PREFETCH_MEN	24	STF	25	STG
26	STS	27	STT	28	LDL
28 ¹	PREFETCH	29	LDQ	29 ¹	PREFETCH_EN
2A	LDL_L	2B	LDQ_L	2C	STL
2D	STQ	2E	STL_C	2F	STQ_C
30	BR	31	FBEQ	32	FBLT
33	FBLE	34	BSR	35	FBNE
36	FBGE	37	FBGT	38	BLBC
39	BEQ	3A	BLT	3B	BLE
3C	BLBS	3D	BNE	3E	BGE
3F	BGT				

¹ PREFETCHx instructions share opcodes with the corresponding load instructions. The PREFETCHx instructions are distinguished by, in each case, the Fa or Ra operand set to 31.

C.7 OpenVMS PALcode Instruction Summary

Table C–9 lists the OpenVMS unprivileged PALcode instructions.

Table C–9 OpenVMS Unprivileged PALcode Instructions

Mnemonic	Opcode	Description
AMOV RM	00.00A1	Atomic move from register to memory
AMOV RR	00.00A0	Atomic move from register to register
BPT	00.0080	Breakpoint
BUGCHK	00.0081	Bugcheck
CHMK	00.0083	Change mode to kernel
CHME	00.0082	Change mode to executive
CHMS	00.0084	Change mode to supervisor
CHMU	00.0085	Change mode to user
CLRFEN	00.00AE	Clear floating-point enable
GENTRAP	00.00AA	Generate software trap
IMB	00.0086	I-stream memory barrier
INSQHIL	00.0087	Insert into longword queue at head interlocked
INSQHILR	00.00A2	Insert into longword queue at head interlocked resident
INSQHIQ	00.0089	Insert into quadword queue at head interlocked
INSQHIQR	00.00A4	Insert into quadword queue at head interlocked resident
INSQTIL	00.0088	Insert into longword queue at tail interlocked
INSQTILR	00.00A3	Insert into longword queue at tail interlocked resident

Table C–9 OpenVMS Unprivileged PALcode Instructions (Continued)

Mnemonic	Opcod	Description
INSQTIQ	00.008A	Insert into quadword queue at tail interlocked
INSQTIQR	00.00A5	Insert into quadword queue at tail interlockedresident
INSQUEL	00.008B	Insert entry into longword queue
INSQUEL/D	00.008D	Insert entry into longword queue deferred
INSQUEQ	00.008C	Insert entry into quadword queue
INSQUEQ/D	00.008E	Insert entry into quadword queue deferred
PROBER	00.008F	Probe for read access
PROBEW	00.0090	Probe for write access
RD_PS	00.0091	Move processor status
READ_UNQ	00.009E	Read unique context
REI	00.0092	Return from exception or interrupt
REMQHIL	00.0093	Remove from longword queue at head interlocked
REMQHILR	00.00A6	Remove from longword queue at head interlocked resident
REMQHIQ	00.0095	Remove from quadword queue at head interlocked
REMQHIQR	00.00A8	Remove from quadword queue at head interlocked resident
REMQTIL	00.0094	Remove from longword queue at tail interlocked
REMQTILR	00.00A7	Remove from longword queue at tail interlocked resident
REMQTIQ	00.0096	Remove from quadword queue at tail interlocked
REMQTIQR	00.00A9	Remove from quadword queue at tail interlocked resident
REMQUEL	00.0097	Remove entry from longword queue
REMQUEL/D	00.0099	Remove entry from longword queue deferred
REMQUEQ	00.0098	Remove entry from quadword queue
REMQUEQ/D	00.009A	Remove entry from quadword queue deferred
RSCC	00.009D	Read system cycle counter
SWASTEN	00.009B	Swap AST enable for current mode
WRITE_UNQ	00.009F	Write unique context
WR_PS_SW	00.009C	Write processor status software field

Table C–10 lists the OpenVMS privileged PALcode instructions.

Table C–10 OpenVMS Privileged PALcode Instructions

Mnemonic	Opcod	Description
CFLUSH	00.0001	Cache flush
CSERVE	00.0009	Console service
DRAINA	00.0002	Drain aborts
HALT	00.0000	Halt processor
LDQP	00.0003	Load quadword physical
MFPR_ASN	00.0006	Move from processor register ASN
MFPR_ESP	00.001E	Move from processor register ESP
MFPR_FEN	00.000B	Move from processor register FEN
MFPR_IPL	00.000E	Move from processor register IPL

Table C–10 OpenVMS Privileged PALcode Instructions (Continued)

Mnemonic	Opcode	Description
MFPR_MCES	00.0010	Move from processor register MCES
MFPR_PCBB	00.0012	Move from processor register PCBB
MFPR_PRBR	00.0013	Move from processor register PRBR
MFPR_PTBR	00.0015	Move from processor register PTBR
MFPR_SCBB	00.0016	Move from processor register SCBB
MFPR_SISR	00.0019	Move from processor register SISR
MFPR_SSP	00.0020	Move from processor register SSP
MFPR_SYSPTBR	00.0032	Move from processor register SYSPTBR
MFPR_TBCHK	00.001A	Move from processor register TBCHK
MFPR_USP	00.0022	Move from processor register USP
MFPT_VIRBND	00.0030	Move from processor register VIRBND
MFPR_VPTB	00.0029	Move from processor register VPTB
MFPR_WHAMI	00.003F	Move from processor register WHAMI
MTPR_ASTEN	00.0026	Move to processor register ASTEN
MTPR_ASTSR	00.0027	Move to processor register ASTSR
MTPR_DATFX	00.002E	Move to processor register DATFX
MTPR_ESP	00.001F	Move to processor register ESP
MTPR_FEN	00.000B	Move to processor register FEN
MTPR_IPIR	00.000D	Move to processor register IPRI
MTPR_IPL	00.000E	Move to processor register IPL
MTPR_MCES	00.0011	Move to processor register MCES
MTPR_PERFMON	00.002B	Move to processor register PERFMON
MTPR_PRBR	00.0014	Move to processor register PRBR
MTPR_SCBB	00.0017	Move to processor register SCBB
MTPR_SIRR	00.0018	Move to processor register SIRR
MTPR_SSP	00.0021	Move to processor register SSP
MTPR_SYSPTBR	00.0033	Move to processor register SYSPTBR
MTPR_TBIA	00.001B	Move to processor register TBIA
MTPR_TBIAP	00.001C	Move to processor register TBIAP
MTPR_TBIS	00.001D	Move to processor register TBIS
MTPR_TBISD	00.0024	Move to processor register TBISD
MTPR_TBISI	00.0025	Move to processor register TBISI
MTPR_USP	00.0023	Move to processor register USP
MTPR_VIRBND	00.0031	Move to processor register VIRBND
MTPR_VPTB	00.002A	Move to processor register VPTB
STQP	00.0004	Store quadword physical
SWPCTX	00.0005	Swap privileged context
SWPPAL	00.000A	Swap PALcode image
WTINT	00.003E	Wait for interrupt

C.8 Tru64 UNIX PALcode Instruction Summary

Table C–11 lists the Tru64 UNIX unprivileged PALcode instructions.

Table C–11 Tru64 UNIX Unprivileged PALcode Instructions

Mnemonic	Opcode	Description
bpt	00.0080	Breakpoint trap
bugchk	00.0081	Bugcheck
callsys	00.0083	System call
clrfen	00.00AE	Clear floating-point enable
gentrap	00.00AA	Generate software trap
imb	00.0086	I-stream memory barrier
rdunique	00.009E	Read unique value
urti	00.0092	Return from user mode trap
wrunique	00.009F	Write unique value

Table C–12 lists the Tru64 UNIX unprivileged PALcode instructions.

Table C–12 Tru64 UNIX Privileged PALcode Instructions

Mnemonic	Opcode	Description
cflush	00.0001	Cache flush
cserve	00.0009	Console service
draina	00.0002	Drain aborts
halt	00.0000	Halt the processor
rdmces	00.0010	Read machine check error summary register
rdps	00.0036	Read processor status
rdusp	00.003A	Read user stack pointer
rdval	00.0032	Read system value
retsys	00.003D	Return from system call
rti	00.003F	Return from trap or interrupt
swpctx	00.0030	Swap privileged context
swpipl	00.0035	Swap interrupt priority level
swppal	00.000A	Swap PALcode image
tbi	00.0033	Translation buffer invalidate
whami	00.003C	Who am I
wrasn	00.002E	Write ASN
wrent	00.0034	Write system entry address
wrfen	00.002B	Write floating-point enable
wripir	00.000D	Write interprocessor interrupt request
wrkgp	00.0037	Write kernel global pointer
wrmces	00.0011	Write machine check error summary register
wrperfmon	00.0039	Performance monitoring function
wrusp	00.0038	Write user stack pointer

Table C–12 Tru64 UNIX Privileged PALcode Instructions (Continued)

Mnemonic	Opcode	Description
wrval	00.0031	Write system value
wrsysptb	00.0014	Write system page table base
wrvirbnd	00.0013	Write virtual address boundary
wrvptptr	00.002D	Write virtual page table pointer
wtint	00.003E	Wait for interrupt

C.9 Alpha Linux PALcode Instruction Summary

Table C–13 lists the Alpha Linux unprivileged PALcode instructions.

Table C–13 Alpha Linux Unprivileged PALcode Instructions

Mnemonic	Opcode	Description
bpt	00.0080	Breakpoint trap
bugchk	00.0081	Bugcheck
callsys	00.0083	System call
clrfen	00.00AE	Clear floating-point enable
gentrap	00.00AA	Generate software trap
imb	00.0086	I-stream memory barrier
rdunique	00.009E	Read unique value
wrunique	00.009F	Write unique value

Table C–14 lists the Alpha Linux privileged PALcode instructions.

Table C–14 Alpha Linux Privileged PALcode Instructions

Mnemonic	Opcode	Description
cflush	00.0001	Cache flush
cserve	00.0009	Console service
draina	00.0002	Drain aborts
halt	00.0000	Halt the processor
rdmces	00.0010	Read machine check error summary register
rdps	00.0036	Read processor status
rdusp	00.003A	Read user stack pointer
rdval	00.0032	Read system value
retsyz	00.003D	Return from system call
rti	00.003F	Return from trap or interrupt
swpctx	00.0030	Swap privileged context
swpipl	00.0035	Swap interrupt priority level
swppal	00.000A	Swap PALcode image
tbi	00.0033	Translation buffer invalidate
whami	00.003C	Who am I
wrent	00.0034	Write system entry address

Table C–14 Alpha Linux Privileged PALcode Instructions (Continued)

Mnemonic	Opcode	Description
wrfen	00.002B	Write floating-point enable
wripir	00.000D	Write interprocessor interrupt request
wrkgp	00.0037	Write kernel global pointer
wrmces	00.0011	Write machine check error summary register
wrperfmon	00.0039	Performance monitoring function
wrusp	00.0038	Write user stack pointer
wrval	00.0031	Write system value
wrsysptb	00.0014	Write system page table base
wrvirbnd	00.0013	Write virtual address boundary
wrvptptr	00.002D	Write virtual page table pointer
wtint	00.003E	Wait for interrupt

C.10 PALcode Opcodes in Numerical Order

Opcodes 00.003816 through 00.003F16 are reserved for processor implementation-specific PALcode instructions. All other opcodes are reserved for use by Compaq.

Table C–15 PALcode Opcodes in Numerical Order

Opcode ₁₆	Opcode ₁₀	OpenVMS	Tru64 UNIX	Alpha Linux
00.0000	00.0000	HALT	halt	halt
00.0001	00.0001	CFLUSH	cflush	cflush
00.0002	00.0002	DRAINA	draina	draina
00.0003	00.0003	LDQP	—	—
00.0004	00.0004	STQP	—	—
00.0005	00.0005	SWPCTX	—	—
00.0006	00.0006	MFPR_ASN	—	—
00.0007	00.0007	MTPR_ASTEN	—	—
00.0008	00.0008	MTPR_ASTR	—	—
00.0009	00.0009	CSERVE	cserve	cserve
00.000A	00.0010	SWPPAL	swppal	swppal
00.000B	00.0011	MFPR_FEN	—	—
00.000C	00.0012	MTPR_FEN	—	—
00.000D	00.0013	MTPR_IPIR	wripir	wripir
00.000E	00.0014	MFPR_IPL	—	—
00.000F	00.0015	MTPR_IPL	—	—
00.0010	00.0016	MFPR_MCES	rdmces	rdmces
00.0011	00.0017	MTPR_MCES	wrmces	wrmces
00.0012	00.0018	MFPR_PCBB	—	—
00.0013	00.0019	MFPR_PRBR	—	—
00.0014	00.0020	MTPR_PRBR	—	—

Table C–15 PALcode Opcodes in Numerical Order (Continued)

Opcode₁₆	Opcode₁₀	OpenVMS	Tru64 UNIX	Alpha Linux
00.0015	00.0021	MFPR_PTBR	—	—
00.0016	00.0022	MFPR_SCBB	—	—
00.0017	00.0023	MTPR_SCBB	—	—
00.0018	00.0024	MTPR_SIRR	—	—
00.0019	00.0025	MFPR_SISR	—	—
00.001A	00.0026	MFPR_TBCHK	—	—
00.001B	00.0027	MTPR_TBIA	—	—
00.001C	00.0028	MTPR_TBIAP	—	—
00.001D	00.0029	MTPR_TBIS	—	—
00.001E	00.0030	MFPR_ESP	—	—
00.001F	00.0031	MTPR_ESP	—	—
00.0020	00.0032	MFPR_SSP	—	—
00.0021	00.0033	MTPR_SSP	—	—
00.0022	00.0034	MFPR_USP	—	—
00.0023	00.0035	MTPR_USP	—	—
00.0024	00.0036	MTPR_TBISD	—	—
00.0025	00.0037	MTPR_TBISI	—	—
00.0026	00.0038	MFPR_ASTEN	—	—
00.0027	00.0039	MFPR_ASTSR	—	—
00.0029	00.0041	MFPR_VPTB	—	—
00.002A	00.0042	MTPR_VPTB	—	—
00.002B	00.0043	MTPR_PERFMON	wrfen	wrfen
00.002D	00.0045	—	wrvptptr	wrvptptr
00.002E	00.0046	MTPR_DATFX	wrasn	—
00.0030	00.0048	—	swpctx	swpctx
00.0031	00.0049	—	wrval	wrval
00.0032	00.0050	—	rdval	rdval
00.0033	00.0051	—	tbi	tbi
00.0034	00.0052	—	wrent	wrent
00.0035	00.0053	—	swpipl	swpipl
00.0036	00.0054	—	rdps	rdps
00.0037	00.0055	—	wrkgp	wrkgp
00.0038	00.0056	—	wrusp	wrusp
00.0039	00.0057	—	wrperfmon	wrperfmon
00.003A	00.0058	—	rdusp	rdusp
00.003C	00.0060	—	whami	whami
00.003D	00.0061	—	retsys	retsys
00.003E	00.0062	WTINT	wtint	wtint
00.003F	00.0063	MFPR_WHAMI	rti	rti

Table C–15 PALcode Opcodes in Numerical Order (Continued)

Opcode₁₆	Opcode₁₀	OpenVMS	Tru64 UNIX	Alpha Linux
00.0080	00.0128	BPT	bpt	bpt
00.0081	00.0129	BUGCHK	bugchk	bugchk
00.0082	00.0130	CHME	—	—
00.0083	00.0131	CHMK	callsys	callsys
00.0084	00.0132	CHMS	—	—
00.0085	00.0133	CHMU	—	—
00.0086	00.0134	IMB	imb	imb
00.0087	00.0135	INSQHIL	—	—
00.0088	00.0136	INSQTIL	—	—
00.0089	00.0137	INSQHIQ	—	—
00.008A	00.0138	INSQTIQ	—	—
00.008B	00.0139	INSQUEL	—	—
00.008C	00.0140	INSQUEQ	—	—
00.008D	00.0141	INSQUEL/D	—	—
00.008E	00.0142	INSQUEQ/D	—	—
00.008F	00.0143	PROBER	—	—
00.0090	00.0144	PROBEW	—	—
00.0091	00.0145	RD_PS	—	—
00.0092	00.0146	REI	urti	—
00.0093	00.0147	REMQHIL	—	—
00.0094	00.0148	REMQTIL	—	—
00.0095	00.0149	REMQHIQ	—	—
00.0096	00.0150	REMQTIQ	—	—
00.0097	00.0151	REMQUEL	—	—
00.0098	00.0152	REMQUEQ	—	—
00.0099	00.0153	REMQUEL/D	—	—
00.009A	00.0154	REMQUEQ/D	—	—
00.009B	00.0155	SWASTEN	—	—
00.009C	00.0156	WR_PS_SW	—	—
00.009D	00.0157	RSCC	—	—
00.009E	00.0158	READ_UNQ	rdunique	rdunique
00.009F	00.0159	WRITE_UNQ	wrunique	wrunique
00.00A0	00.0160	AMOVRR	—	—
00.00A1	00.0161	AMOVRM	—	—
00.00A2	00.0162	INSQHILR	—	—
00.00A3	00.0163	INSQTILR	—	—
00.00A4	00.0164	INSQHIQR	—	—
00.00A5	00.0165	INSQTIQR	—	—
00.00A6	00.0166	REMQHILR	—	—

Table C–15 PALcode Opcodes in Numerical Order (Continued)

Opcode ₁₆	Opcode ₁₀	OpenVMS	Tru64 UNIX	Alpha Linux
00.00A7	00.0167	REMQTILR	—	—
00.00A8	00.0168	REMQHIQR	—	—
00.00A9	00.0169	REMQTIQR	—	—
00.00AA	00.0170	GENTRAP	gentrap	gentrap
00.00AB	00.0171	—	—	—
00.00AC	00.0172	—	—	—
00.00AD	00.0173	—	—	—
00.00AE	00.0174	CLRFEN	clrfen	clrfen

C.11 Required PALcode Opcodes

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 C–16: Required PALcode Opcodes

Mnemonic	Type	Opcode
DRAINA	Privileged	00.0002
HALT	Privileged	00.0000
IMB	Unprivileged	00.0086

C.12 Opcodes Reserved to PALcode

The opcodes listed in Table C–17 are reserved for use in implementing PALcode.

Table C–17: Opcodes Reserved for PALcode

Mnemonic		Mnemonic		Mnemonic	
PAL19	19	PAL1B	1B	PAL1D	1D
PAL1E	1E	PAL1F	1F		

C.13 Opcodes Reserved to Compaq

The opcodes listed in Table C–18 are reserved to Compaq.

Table C–18: Opcodes Reserved for Compaq

Mnemonic		Mnemonic		Mnemonic	
OPC01	01	OPC02	02	OPC03	03
OPC04	04	OPC05	05	OPC06	06
OPC07	07				

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

Table C–19 shows the 7-bit ASCII character set and the corresponding hexadecimal value for each character.

Table C–19: ASCII Character Set

Char	Hex Code	Char	Hex Code	Char	Hex Code	Char	Hex Code
NUL	0	SP	20	@	40	'	60
SQH	1	!	21	A	41	a	61
STX	2	"	22	B	42	b	62
ETX	3	#	23	C	43	c	63
EOT	4	\$	24	D	44	d	64
ENQ	5	%	25	E	45	e	65
ACK	6	&	26	F	46	f	66
BEL	7	'	27	G	47	g	67
BS	8	(28	H	48	h	68
HT	9)	29	I	49	i	69
LF	A	*	2A	J	4A	j	6A
VT	B	+	2B	K	4B	k	6B
FF	C	,	2C	L	4C	l	6C
CR	D	-	2D	M	4D	m	6D
SO	E	.	2E	N	4E	n	6E
SI	F	/	2F	O	4F	o	6F
DLE	10	0	30	P	50	p	70
DC1	11	1	31	Q	51	q	71
DC2	12	2	32	R	52	r	72
DC3	13	3	33	S	53	s	73
DC4	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[5B	{	7B
FS	1C	<	3C	\	5C		7C
GS	1D	=	3D]	5D	}	7D
RS	1E	>	3E	^	5E	~	7E
US	1F	?	3F	_	5F	DEL	7F

Registered System and Processor Identifiers

This appendix contains a table of the processor type assignments, PALcode implementation information, and the architecture mask (AMASK) and implementation value (IMPLVER) assignments.

D.1 Processor Type Assignments

The following processor types are defined.

Table D–1: Processor Type Assignments

Major Type	Minor Type
1 = EV3	
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 (Pass 1) 1 = Pass 2, 2.2 (rev BA, CA) 2 = Pass 2.3 (rev DA, EA)

Table D–1: Processor Type Assignments (Continued)

Major Type	Minor Type
	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 = 21264/EV6	0 = Reserved
	1 = Pass 1
	2 = Pass 2, 2.1
	3 = Pass 2.2
	4 = Pass 2.3
	5 = Pass 3
	6 = Pass 2.4
	7 = Pass 2.5
9 = PCA56 (21164PC)	0 = Reserved
	1 = Pass 1
10 = PCA57	0 = Reserved
	1 = Pass 1
11 = 21264/EV67	0 = Reserved
	1 = Pass 1
	2 = Pass 2.1
	3 = Pass 2.2
	4 = Pass 2.1.1
	5 = Pass 2.2.1
	6 = Pass 2.3 and Pass 2.4
	7 = Pass 2.1.2
	8 = Pass 2.2.2
	9 = Pass 2.2.3 and Pass 2.2.5

Table D–1: Processor Type Assignments (Continued)

Major Type	Minor Type
	10 = Pass 2.2.4
	11 = Pass 2.5
	12 = Pass 2.4.1
	13 = Pass 2.5.1
	14 = Pass 2.6
12 = 21264/EV68CB	0 = Reserved
	1 = Pass 1
	2 = Pass 2 and Pass 2.1
	3 = Pass 2.2 and Pass 2.3
	4 = Pass 3 and Pass 3.1
	5 = Pass 2.4
	6 = Pass 4
21264/EV68DC	0 = Reserved
	1 = Pass 1
	2 = Pass 2
	3 = Pass 2.3.1
	4 = Pass 3 and Pass 3.1
	5 = Pass 2.4
	6 = Pass 4
13 = 21264/EV68A	0 = Reserved
	1 = Pass 1
	2 = Pass 2
	3 = Pass 2.1 and Pass 2.1A and Pass 3
	4 = Pass 2.2
14 = 21264/EV68CX	0 = Reserved
	1 = Pass 1
15 = 21364/EV7	0 = Reserved
	1 = Pass 1
16 = 21364/EV79	0 = Reserved
	1 = Pass 1
17 = 21264/EV69A	0 = Reserved
	1 = Pass 1

For OpenVMS, Tru64 UNIX, and Alpha Linux, the processor types are stored in the Per-CPU Slot Table (SLOT[176]), pointed to by HWRPB[160].

D.2 PALcode Variation Assignments

The PALcode variation assignments are as follows:

Table D–2: PALcode Variation Assignments

Token	PALcode Type	Summary
0	Console	N/A
1	OpenVMS	Section 27.4
2	Tru64 UNIX and Alpha Linux	Section 27.4
3–127	Reserved to Compaq	
128–255	Reserved to non-Compaq	

D.3 Architecture Mask and Implementation Values

The following bits are defined for the AMASK instruction.

Table D–3 AMASK Bit Assignments

Bit	Meaning
0	Support for the byte/word extension (BWX) The instructions that comprise the BWX extension are LDBU, LDWU, SEXTB, SEXTW, STB, and STW.
1	Support for the square-root and floating-point convert extension (FIX) The instructions that comprise the FIX extension are FTOIS, FTOIT, ITOFF, ITOFS, ITOFT, SQRTF, SQRTG, SQRTS, and SQRTT.
2	Support for the count extension (CIX) The instructions that comprise the CIX extension are CTLZ, CTPOP, and CTTZ.
8	Support for the multimedia extension (MVI) The instructions that comprise the MVI extension are MAXSB8, MAXSW4, MAXUB8, MAXUW4, MINSB8, MINSW4, MINUB8, MINUW4, PERR, PKLB, PKWB, UNPKBL, and UNPKBW.
9	Support for precise arithmetic trap reporting in hardware. The trap PC is the same as the instruction PC after the trapping instruction is executed.
10	Not available.
11	Not available.
12	Support for prefetch with modify intent to improve the performance of the first attempt to acquire a lock.

The following values are defined for the IMPLVER instruction.

Table D-4: IMPLVER Value Assignments

Value	Meaning
0	21064 (EV4) 21064A (EV45) 21066A/21068A (LCA45)
1	21164 (EV5) 21164A (EV56) 21164PC (PCA56)
2	21264/EV6 21264/EV67 21264/EV68x
3	21364/EV7 21364/EV79

Waivers and Implementation-Dependent Functionality

This appendix describes waivers to the Alpha architecture and functionality that is specific to particular hardware implementations.

E.1 Waivers

The following waivers have been passed for the Alpha architecture.

E.1.1 21064, 21066, and 21068 IEEE Divide Instruction Violation

The 21064, 21066, and 21068 CPUs violate the architected handling of IEEE divide instructions DIVS and DIVT with respect to reporting Inexact Result exceptions.

Note:

The 21064A, 21066A, and 21068A CPUs are compliant and require no waiver. The 21164 is also compliant.

As specified by the architecture, floating-point exceptions generated by the CPU are recorded in two places for all IEEE floating-point instructions:

1. If an exception is detected and the corresponding trap is enabled (such as ADD/U for underflow), the CPU initiates a trap and records the exception in the exception summary register (EXC_SUM).
2. The exceptions are also recorded as flags that can be tested in the floating-point control register (FPCR). The FPCR can only be accessed with MTPR/MFPR instructions and an explicit MT_FPCR is required to clear the FPCR. The FPCR is updated irrespective of whether the trap is enabled or not.

The 21064, 21066, and 21068 implementations differ from the above specification in handling the Inexact condition for the IEEE DIVS and DIVT instructions in two ways:

1. The DIVS and DIVT instructions with the /Inexact modifier trap unconditionally and report the INE exception in the EXC_SUM register (except for NaN, infinity, and denormal inputs that result in INVs). This allows for a software calculation to determine the correct INE status.

2. The FPCR <INE> bit is *never* set by DIVS or DIVT. This is because the 21064, 21066, and 21068 do not include hardware to determine that particular exactness.

E.1.2 21064, 21066, and 21068 Write Buffer Violation

The 21064, 21066, and 21068 CPUs can be made to violate the architecture by, under one contrived case, indefinitely delaying a buffered offchip write.

Note:

The 21064A, 21066A, and 21068A CPUs are compliant and require no waiver. The 21164 is also compliant.

The CPUs in violation can send a buffered write offchip when one of the following conditions is met:

1. The write buffer contains at least two valid entries.
2. The write buffer contains one valid entry and 256 cycles have elapsed since the execution of the last write.
3. The write buffer contains an MB or STx_C instruction.
4. A load miss hits an entry in the write buffer.

The write can be delayed indefinitely under condition 2 above, when there is an indefinite stream of writes to addresses within the same aligned 32-byte write buffer block.

E.1.3 21264 LDx_L/STx_C with WH64 Violation

The 21264 violates the architected relationship between the LDx_L and STx_C instructions when an intervening WH64 instruction is executed.

As specified in Section 4.2.4:

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.

The 21264 varies from that description, with regard to the WH64 instruction, as follows:

If any other memory access (ECB, LDx, LDQ_U, STx, STQ_U) 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 WH64 memory access is executed on any given 21264 processor between the LDx_L and STx_C, and:

- The WH64 access is to the same aligned 64-byte block that STx_C is accessing, and
- No CALL_PAL REI, rei, or rfe instruction has been executed since the most-recent LDx_L (ensuring that the sequence cannot occur as the result of unfortunate coincidences with interrupts)

then, the load-locked/store-conditional sequence may sometimes fail when it would otherwise succeed and sometimes succeed when it otherwise would fail; hence no useful program should do this.

E.1.4 21164, 21164A, and 21164PC Operation with RPCC Instruction

The 21164, 21164A, and 21164PC do not fully implement the following specified operation regarding the Rb operand of the RPCC instruction, as defined in Section 4.11.8:

"RPCC does not read the Processor Cycle Counter (PCC) any earlier than the generation of a result by the nearest preceding instruction that modifies register Rb. If R31 is used as the Rb operand, the PCC need not wait for any preceding computation."

Rather, the waived CPUs wait only for the issue of all preceding instructions, including the instruction that modifies register Rb; they do not wait for the generation of the result.

For example, the following code reads the processor cycle counter of a waived CPU without waiting for the multiply to generate a result in register R18:

```
MULQ R16, R17, R18
RPCC R0, R18
```

However, the following sequence waits for the multiply to complete because it waits for the BIS instruction to issue and the BIS does not issue until the multiply generates a result in register R18:

```
MULQ R16, R17, R18
BIS R31, R18, R18
RPCC R0, R18
```

E.1.5 21264/EV6 Behavior on LD_x_L/ST_x_C Synchronization

Passes 2.3 and 3.2 of the 21264/EV6 can exhibit behavior on LD_x_L/ST_x_C synchronization sequences that may be interpreted to be inconsistent with that specified in Sections 4.2.4 and 4.2.5. The waived CPUs behave correctly for LD_x_L/ST_x_C sequences that follow the guidelines in those sections.

For some ill-formed sequences, that is, code sequences that do not follow the guidelines specified in those sections, it may be possible for the waived CPUs to succeed a ST_x_C even though another processor obtained the lock flag between a LD_x_L and the ST_x_C. This behavior might occur if there is a taken branch, JSR, or jump between the LD_x_L and the ST_x_C.

Consider the following (attempted) synchronization sequence:

```
start:
    LDA R2, 1(R31)
    LDQ_L R0, 0(R1)
    BEQ R0, doit
lazy:
    LDQ R0, 0(R1)
    BNE R0, lazy
    BR start
doit:
    STQ_C R2, 0(R1)
    BEQ R2, start
```

Section 4.2.4 states that the "sequence above may always fail on some implementations; hence, no useful program should do this" because there is a taken branch between the LDQ_L and the STQ_C. The waived CPUs may in some cases succeed the STQ_C. In some cases, the waived CPUs (incorrectly) succeed the STQ_C even though another processor steals the lock.

The following sequence of events might cause this behavior:

1. The block is in the cache and the lock is not set.
2. The LDQ_L issues, returning a value of zero.
3. Another processor writes a one to the lock (via a STQ_C, most likely). This evicts the block containing 0(R1) from the cache.
4. The doit branch mispredicts. The waived CPU speculatively issues the LDQ on the fall-through path under the doit branch. This load reloads the cache block containing 0(R1) into the cache.
5. The waived CPU detects the doit branch mispredict and squashes the LDQ on the fall-through path of the doit branch.
6. The STQ_C finds the block containing 0(R1) in the cache and succeeds (incorrectly), basically because the CPU assumes that if the block is in the cache it must still have the lock.

If, instead, the code is structured to be compliant with the guidelines, as follows:

```
start:
    LDA R2, 1(R31)
    LDQ_L R0, 0(R1)
    BNE R0, lazy
    STQ_C R2, 0(R1)
    BEQ R2, start
    BR done
lazy:
    LDQ R0, 0(R1)
    BNE R0, lazy
    BR start
done:
```

There is no taken-branch between the LDx_L and STx_C. This sequence works correctly on all passes of the 21264. The scenario of incorrectly succeeding the STQ_C cannot happen because (4) cannot happen — branches after a LDQ_L cannot predict taken while the lock is active.

E.1.6 21264/EV6 and 21264/EV67 Prefetch and Lock Behavior

For 21264/EV6 and 21264/EV67 processors, a cache block prefetch within a dynamic 80-instruction window before a LDx_L can cause the subsequent STx_C to succeed incorrectly if all three instructions reference the same 64-byte cache block. The incorrect operation cannot occur in subsequent processors. Subsequent processors do not have the waived behavior and can correctly prefetch locked memory blocks.

The AMASK instruction can be used to test for the waived condition. On Alpha implementations that are not waived, AMASK clears feature mask bit 12; for those implementations that are waived, AMASK does not clear that bit. See Section 5.5.4 for using the AMASK instruction and feature mask bit 12 with prefetching locked memory blocks.

Implementation Note:

With a waived processor, the branch based on examining AMASK<12> could be mispredicted and a cache block prefetch described above could be speculatively executed. Such a case does not expose the waived condition; the condition cannot occur if the prefetch is only a mispredicated path.

E.2 Implementation-Specific Functionality

The following functionality, although a documented part of the Alpha architecture, is implemented in a manner that is specific to the particular hardware implementation.

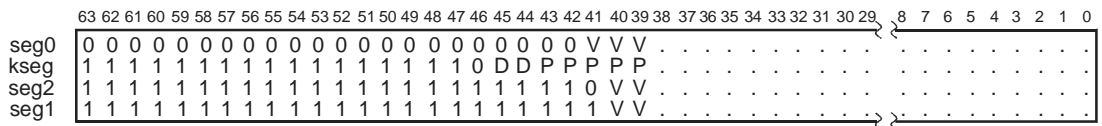
E.2.1 Enlarging the Tru64 UNIX kseg Region

When implemented, the kseg region of virtual memory must be able to map all of physical memory. That requirement is not met on a Tru64 UNIX system with the following characteristics:

- An 8KB page size
- A physical address space that is larger than 41 bits

To meet that requirement, in an implementation-dependent manner, the kseg region can be enlarged over the size of the segx regions. No changes are made to the segx or the PTE formats in implementing this enlargement. As a side-effect of the mixed-size regions, a new but currently unused memory segment, seg2, is created.

For example, a 43-bit segx region and 48-bit kseg region, as implemented for the 21364 with those characteristics, could have the following address space partitions:



D = Do not care, not used for the 21364
P = Physical (and virtual) addresses that are used in kseg on the 21364
V = Virtual address bits that are used

In this example of mixed-size virtual address space, segx addresses are sign-extended from bit <42>, while kseg addresses are sign-extended from bit <47>. In tabular form, the regions are as follows:

Single-size 43-bit regions:

- seg0<42:41> = 0x, <42> = 0 sign-extended to <63>
- kseg<42:41> = 10, <42> = 1 sign-extended to <63>
- seg1<42:41> = 11, <42> = 1 sign-extended to <63>

Mixed-size 48-bit kseg/43-bit segx regions:

seg0<42:41> = 0x, <42> = 0 sign-extended to <63>

kseg<47:46> = 10, <47> = 1 sign-extended to <63>

seg1<42:41> = 11, <42> = 1 sign-extended to <63>, with <41> = 1

seg2<42:41> = 10, <42> = 1 sign-extended to <63>, with <41> = 0

Possible 21364 Virtual Address (segx) Anomaly

The 21364 implementation of mixed-mode segx virtual addressing is as follows:

1. The hardware checks that bits VA<63:48> are a sign-extension of VA<47> on every access, and the PALcode checks that bits VA<47:43> are a sign-extension of VA<42> after the access incurs a TBMISSE.
2. When a TBMISSE occurs, the hardware cannot construct the VA of the lowest-level PTE entry as described in Section 17.6.2 . For performance reasons, the VA that the hardware can construct is used, rather than having the PALcode calculate the VPTE in the TBMISSE routine. Only bits <47:43> of this VA can be pre-specified by PALcode. To avoid creating aliases to proper 43-bit segx addresses, PALcode must pre-specify bits<47:43> so that they *cannot* be a proper sign-extension of VA<42>. It is this VA that is placed in the TB, mapped to the lowest level PTE entry. This entry must exist in the TB to allow virtual access of the PTE. (Note that PALcode access of the resulting virtual address will not fault. Although bits VA<47:43> are not a proper sign-extension of VA<42>, they are not checked by the hardware.)

It is possible for a kernel-mode access to a virtual address that is properly extended from VA<47>, but with VA<47:43> not properly sign-extended from VA<42>, to erroneously match a TB entry for the virtual address of a lowest-level PTE entry. The incorrectly sign-extended address will be translated instead of correctly being detected. (The access has to be kernel mode because the protection on the PTE in the TB does not allow user-mode access.)

E.2.2 Reduced Page Table (RPT) Mode in the 21364

When a TBMISSE occurs on a VA in the RPT region, the 21364 calculates the virtual address of the level 2 PTE (VPTE) as follows:

$$\begin{aligned} \text{VPTE}<63:42> &\leftarrow \text{SEXT}(\text{VPTB}<47:42>) \\ \text{VPTE}<41:29> &\leftarrow \text{SEXT}(\text{VA}<47:42> \& 30_{16}) = 0000000010000_2 \\ \text{VPTE}<28:16> &\leftarrow \text{SEXT}(\text{VA}<47:42> \& 0F_{16}) = \text{VA}<45:42> \\ \text{VPTE}<15:3> &\leftarrow \text{VA}<41:29> \\ \text{VPTE}<02:0> &\leftarrow 0 \end{aligned}$$

For performance reasons, it is desirable that a single TB miss flow execute without regard to whether the missing VA is mapped by two levels (RPT) or three levels of page table. It is further desirable that a single TB miss flow use the VPTE address as constructed by the 21364, rather than calculating it in PALcode.

The double TB miss flow (entered when the VPTE access causes a TB miss) physically walks the page tables to translate the VPTE address to the physical page that contains the PTE. This translation is put in the TB and the VPTE access is retried. Note that the original missing VA is never translated by a physical walk of the page tables.

The same double TB miss flow is entered whether or not the original missing VA resided in the RPT region. The physical walk of the page tables relies on self-map entries in the page table to "back up" one or more levels, so the final PTE obtained is the PTE for the lowest level of page table, mapping the original missing VA. This walk also requires that the index fields for the intermediate page table levels, while they might be shifted (that is, the original Level1 index might become a Level2 index), must be left intact during the transformation from VA to VPTE. However, as shown above, when the original missing VA is in the RPT region, the 21364 separates VA<47:42> across two index fields in the VPTE address, which breaks the indexing mechanism. So, a simple physical walk of the page tables cannot translate this VPTE address.

There are (at least) two remedies to this problem: 1) make the double TB miss flow RPT-aware, translating the VPTE address in an alternate manner when the original missing VA resides in the RPT region; or 2) create a structure such that the double TB miss flow can continue to do a three-level physical walk without being RPT-aware.

The PALcode assumes the second remedy. The construction of the page table is changed such that VA<45:42> is the complete Level1 index into an alternate Level1 page table (the RPT Level1 page table) that is used only for the RPT region. Because the PALcode never translates the original missing VA by a physical walk of the page tables, the PTEs in the normal Level1 page table, indexed by VA<47:42> = 01xxxx₂, can be used for a different purpose as described below. Note that with this change, the algorithm in Sections 11.8.3.1 (OpenVMS), 17.6.3.1 (Tru64 UNIX), and 22.6.3.1 (Alpha Linux) cannot be used to translate the original missing VA.

The 21364 produces a VPTE address with Level2 index bits (VPTE<41:29>) of 010000₂, which is the index of the first entry in the RPT region of the normal Level1 page table. This PTE can now be used to map the RPT Level1 page table. The rest of the PTEs in that region are left unused.

Using page tables constructed in this manner, the double TB miss flow translation of the VPTE address can proceed whether or not the original missing VA is in the RPT region.

E.2.3 21064/21066/21068 Performance Monitoring

Note:

All functions, arguments, and descriptions in this section apply to the 21064/21064A, 21066/21066A, and 21068/21068A.

PALcode instructions control the 21064/21066/21068 onchip performance counters. For OpenVMS, the instruction is MTPR_PERFMON; for Tru64 UNIX and Alpha Linux, the instruction is wrperfmon.

The instruction arguments and results are described in the following sections. The scratch register usage is operating system specific.

Two onchip counters count events. The bit width of the counters (8, 12, or 16 bits) can be selected and the event that they count can be switched among a number of available events. One possible event is an "external" event. For example, the processor board can supply an event that causes the counter to increment. In this manner, offchip events can be counted.

The two counters can be switched independently. There is no hardware support for reading, writing, or resetting the counters. The only way to monitor the counters is to enable them to cause an interrupt on overflow.

The performance monitor functions, described in Section E.2.3.2, can provide the following, depending on implementation:

- Enable the performance counters to interrupt and trap into the performance monitoring vector in the operating system.
- Disable the performance counter from interrupting. This does not necessarily mean that the counters will stop counting.
- Select which events will be monitored and set the width of the two counters.
- In the case of OpenVMS, Tru64 UNIX, and Alpha Linux, implementations can choose to monitor selected processes. If that option is selected, the PME bit in the PCB controls the enabling of the counters. Since the counters cannot be read/written/reset, if more than one process is being monitored, the rounding error may become significant.

E.2.3.1 21064/21066/21068 Performance Monitor Interrupt Mechanism

The performance monitoring interrupt mechanism varies according to the particular operating system.

For the OpenVMS Operating System

When a counter overflows and interrupt enabling conditions are correct, the counter causes an interrupt to PALcode. The PALcode builds an appropriate stack frame. The PALcode then dispatches in the form of an exception (not in the form of an interrupt) to the operating system by vectoring to the SCB performance monitor entry point through SCBB+650 (HWSCB\$Q_PERF_MONITOR), at IPL 29, in kernel mode.

Two interrupts are generated if both counters overflow. For each interrupt, the status of each counter overflow is indicated by register R4:

R4 = 0 if performance counter 0 caused the interrupt
R4 = 1 if performance counter 1 caused the interrupt

When the interrupt is taken, the PC is saved on the stack frame as the old PC.

For the Tru64 UNIX and Alpha Linux Operating Systems

When a counter overflows and interrupt enabling conditions are correct, the counter causes an interrupt to PALcode. The PALcode builds an appropriate stack frame and dispatches to the operating system by vectoring to the interrupt entry point entINT, at IPL 6, in kernel mode.

Two interrupts are generated if both counters overflow. For each interrupt, registers a0..a2 are as follows:

a0 = osfint\$c_perf (4)
a1 = scb\$v_perfmon (650)
a2 = 0 if performance counter 0 caused the interrupt
a2 = 1 if performance counter 1 caused the interrupt

When the interrupt is taken, the PC is saved on the stack frame as the old PC.

E.2.3.2 Functions and Arguments for the 21064/21066/21068

The functions execute on a single (the current running) processor only and are described in Table E-1.

- The OpenVMS MTPR_PERFMON instruction is called with a function code in R16, a function-specific argument in R17, and status is returned in R0.
- The Tru64 UNIX and Alpha Linux wrperfmon instruction is called with a function code in a0, a function specific argument in a1, and status is returned in v0.

Table E-1 21064/21066/21068 Performance Monitoring Functions

Function	Register Usage	Comments
Enable performance monitoring		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 1	Function code
	a1 = 0	Argument
Output:	v0 = 1	Success
	v0 = 0	Failure (not generated)
OpenVMS		
Input:	R16 = 1	Function code
	R17 = 0	Argument
Output:	R0 = 1	Success
	R0 = 0	Failure (not generated)
Disable performance monitoring		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 0	Function code
	a1 = 0	Argument
Output:	v0 = 1	Success
	v0 = 0	Failure (not generated)
OpenVMS		
Input:	R16 = 0	Function code
	R17 = 0	Argument
Output:	R0 = 1	Success
	R0 = 0	Failure (not generated)

Table E–1 21064/21066/21068 Performance Monitoring Functions (Continued)

Function	Register Usage	Comments
Select desired events (mux_ctl)		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 2 a1 = mux_ctl	Function code <i>mux_ctl</i> is the exact contents of those fields from the ICCSR register, in write format, described in Table E–2.
Output:	v0 = 1 v0 = 0	Success Failure (not generated)
OpenVMS		
Input:	R16 = 2 R17 = mux_ctl	Function code <i>mux_ctl</i> is the exact contents of those fields from the ICCSR register, in write format, described in Table E–2.
Output:	R0 = 1 R0 = 0	Success Failure (not generated)
Select performance monitoring options		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 3 a1 = opt	Function code Function argument <i>opt</i> is: <0> = log all processes if set <1> = log only selected if set
Output:	v0 = 1 v0 = 0	Success Failure (not generated)
OpenVMS		
Input:	R16 = 3 R17 = opt	Function code Function argument <i>opt</i> is: <0> = log all processes if set <1> = log only selected if set
Output:	R0 = 1 R0 = 0	Success Failure (not generated)

Table E–2 21064/21066/21068 MUX Control Fields in ICCSR Register

Bits	Option	Description																																		
34:32	PCMUX1	Event selection, counter 1: <table border="1"> <thead> <tr> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr><td>0</td><td>Total D-cache misses</td></tr> <tr><td>1</td><td>Total I-cache misses</td></tr> <tr><td>2</td><td>Cycles of dual issue</td></tr> <tr><td>3</td><td>Branch mispredicts (conditional, JSR, HW_REI)</td></tr> <tr><td>4</td><td>FP operate instructions (not BR, LOAD, STORE)</td></tr> <tr><td>5</td><td>Integer operates (including LDA, LDAH into R0–R30)</td></tr> <tr><td>6</td><td>Total store instructions</td></tr> <tr><td>7</td><td>External events supplied by pin</td></tr> </tbody> </table>	Value	Description	0	Total D-cache misses	1	Total I-cache misses	2	Cycles of dual issue	3	Branch mispredicts (conditional, JSR, HW_REI)	4	FP operate instructions (not BR, LOAD, STORE)	5	Integer operates (including LDA, LDAH into R0–R30)	6	Total store instructions	7	External events supplied by pin																
Value	Description																																			
0	Total D-cache misses																																			
1	Total I-cache misses																																			
2	Cycles of dual issue																																			
3	Branch mispredicts (conditional, JSR, HW_REI)																																			
4	FP operate instructions (not BR, LOAD, STORE)																																			
5	Integer operates (including LDA, LDAH into R0–R30)																																			
6	Total store instructions																																			
7	External events supplied by pin																																			
11:8	PCMUX0	Event selection, counter 0: <table border="1"> <thead> <tr> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr><td>0</td><td>Total issues divided by 2</td></tr> <tr><td>1</td><td>Unused</td></tr> <tr><td>2</td><td>Nothing issued, no valid I-stream data</td></tr> <tr><td>3</td><td>Unused</td></tr> <tr><td>4</td><td>All load instructions</td></tr> <tr><td>5</td><td>Unused</td></tr> <tr><td>6</td><td>Nothing issued, resource conflict</td></tr> <tr><td>7</td><td>Unused</td></tr> <tr><td>8</td><td>All branches (conditional, unconditional, JSR, HW_REI)</td></tr> <tr><td>9</td><td>Unused</td></tr> <tr><td>10</td><td>Total cycles</td></tr> <tr><td>11</td><td>Cycles while in PALcode environment</td></tr> <tr><td>12</td><td>Total nonissues divided by 2</td></tr> <tr><td>13</td><td>Unused</td></tr> <tr><td>14</td><td>External event supplied by pin.</td></tr> <tr><td>15</td><td>Unused</td></tr> </tbody> </table>	Value	Description	0	Total issues divided by 2	1	Unused	2	Nothing issued, no valid I-stream data	3	Unused	4	All load instructions	5	Unused	6	Nothing issued, resource conflict	7	Unused	8	All branches (conditional, unconditional, JSR, HW_REI)	9	Unused	10	Total cycles	11	Cycles while in PALcode environment	12	Total nonissues divided by 2	13	Unused	14	External event supplied by pin.	15	Unused
Value	Description																																			
0	Total issues divided by 2																																			
1	Unused																																			
2	Nothing issued, no valid I-stream data																																			
3	Unused																																			
4	All load instructions																																			
5	Unused																																			
6	Nothing issued, resource conflict																																			
7	Unused																																			
8	All branches (conditional, unconditional, JSR, HW_REI)																																			
9	Unused																																			
10	Total cycles																																			
11	Cycles while in PALcode environment																																			
12	Total nonissues divided by 2																																			
13	Unused																																			
14	External event supplied by pin.																																			
15	Unused																																			
3	PC0	Frequency setting, counter 0: <table border="1"> <thead> <tr> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr><td>0</td><td>2**16 (65536) events per interrupt</td></tr> <tr><td>1</td><td>2**12 (4096) events per interrupt</td></tr> </tbody> </table>	Value	Description	0	2**16 (65536) events per interrupt	1	2**12 (4096) events per interrupt																												
Value	Description																																			
0	2**16 (65536) events per interrupt																																			
1	2**12 (4096) events per interrupt																																			
0	PC1	Frequency setting, counter 1: <table border="1"> <thead> <tr> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr><td>0</td><td>2**12 (4096) events per interrupt</td></tr> <tr><td>1</td><td>2**8 (256) events per interrupt</td></tr> </tbody> </table>	Value	Description	0	2**12 (4096) events per interrupt	1	2**8 (256) events per interrupt																												
Value	Description																																			
0	2**12 (4096) events per interrupt																																			
1	2**8 (256) events per interrupt																																			

E.2.4 21164/21164PC Performance Monitoring

Unless otherwise stated, the term "21164" in this section means implementations of the 21164 at all frequencies.

PALcode instructions control the 21164/21164PC onchip performance counters. For OpenVMS, the instruction is MTPR_PERFMON; for Tru64 UNIX and Alpha Linux, the instruction is wrperfmon.

The instruction arguments and results are described in the following sections. The scratch register usage is operating system specific.

Three onchip counters count events. Counters 0 and 1 are 16-bit counters; counter 2 is a 14-bit counter. Each counter can be individually programmed. Counters can be read and written and are not required to interrupt. The counters can be collectively restricted according to the processor mode.

Processes can be selectively monitored with the PME bit.

E.2.4.1 Performance Monitor Interrupt Mechanism

The performance monitoring interrupt mechanism varies according to the particular operating system.

For the OpenVMS Operating System

When a counter overflows and interrupt enabling conditions are correct, the counter causes an interrupt to PALcode. The PALcode builds an appropriate stack frame. The PALcode then dispatches in the form of an exception (not in the form of an interrupt) to the operating system by vectoring to the SCB performance monitor entry point through SCBB+650 (HWSCB\$Q_PERF_MONITOR), at IPL 29, in kernel mode.

An interrupt is generated for each counter overflow. For each interrupt, the status of each counter overflow is indicated by register R4:

R4 = 0 if performance counter 0 caused the interrupt
R4 = 1 if performance counter 1 caused the interrupt
R4 = 2 if performance counter 2 caused the interrupt

When the interrupt is taken, the PC is saved on the stack frame as the old PC.

For the Tru64 UNIX and ALpha Linux Operating Systems

When a counter overflows and interrupt enabling conditions are correct, the counter causes an interrupt to PALcode. The PALcode builds an appropriate stack frame and dispatches to the operating system by vectoring to the interrupt entry point entINT, at IPL 6, in kernel mode.

An interrupt is generated for each counter overflow. For each interrupt, registers a0..a2 are as follows:

a0 = osfint\$c_perf (4)
a1 = scb\$v_perfmon (650)
a2 = 0 if performance counter 0 caused the interrupt
a2 = 1 if performance counter 1 caused the interrupt

E.2.4.2 Functions and Arguments

The functions execute only on a single (the current running) processor and are described in Table E-3.

The OpenVMS MTPR_PERFMON instruction is called with a function code in R16, a function-specific argument in R17, and status is returned in R0.

The Tru64 UNIX and Alpha Linux wrperfmon instruction is called with a function code in a0, a function specific argument in a1, and status is returned in v0.

Table E-3 Performance Monitoring Functions

Function	Register Usage	Comments
Enable performance monitoring; do not reset counters		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 1	Function code value
	a1 = arg	Argument from Table E-4
Output:	v0 = 1	Success
	v0 = 0	Failure (not generated)
OpenVMS		
Input:	R16 = 1	Function code value
	R17 = arg	Argument from Table E-4
Output:	R0 = 1	Success
	R0 = 0	Failure (not generated)
Enable performance monitoring; start the counters from zero		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 7	Function code value
	a1 = arg	Argument from Table E-4
Output:	v0 = 1	Success
	v0 = 0	Failure (not generated)
OpenVMS		
Input:	R16 = 7	Function code value
	R17 = arg	Argument from Table E-4
Output:	R0 = 1	Success
	R0 = 0	Failure (not generated)

Table E-3 Performance Monitoring Functions (Continued)

Function	Register Usage	Comments
Disable performance monitoring; do not reset counters		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 0	Function code value
	a1 = arg	Argument from Table E-5
Output:	v0 = 1	Success
	v0 = 0	Failure (not generated)
OpenVMS		
Input:	R16 = 0	Function code value
	R17 = arg	Argument from Table E-5
Output:	R0 = 1	Success
	R0 = 0	Failure (not generated)
Select desired events (MUX_SELECT)		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 2	Function code value
	a1 = arg	Argument from Table E-6 or E-7
Output:	v0 = 1	Success
	v0 = 0	Failure (not generated)
OpenVMS		
Input:	R16 = 2	Function code value
	R17 = arg	Argument from Table E-6 or E-7
Output:	R0 = 1	Success
	R0 = 0	Failure (not generated)
Select Processor Mode options		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 3	Function code value
	a1 = arg	Argument from Table E-8
Output:	v0 = 1	Success
	v0 = 0	Failure (not generated)
OpenVMS		
Input:	R16 = 3	Function code value
	R17 = arg	Argument from Table E-8
Output:	R0 = 1	Success
	R0 = 0	Failure (not generated)

Table E–3 Performance Monitoring Functions (Continued)

Function	Register Usage	Comments
Select interrupt frequencies		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 4	Function code value
	a1 = arg	Argument from Table E–9
Output:	v0 = 1	Success
	v0 = 0	Failure (not generated)
OpenVMS		
Input:	R16 = 4	Function code value
	R17 = arg	Argument from Table E–9
Output:	R0 = 1	Success
	R0 = 0	Failure (not generated)
Read the counters		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 5	Function code value
	a1 = arg	Argument from Table E–10
Output:	v0 = val	Return value from Table E–10
OpenVMS		
Input:	R16 = 5	Function code value
	R17 = arg	Argument from Table E–10
Output:	R0 = val	Return value from Table E–10
Write the counters		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 6	Function code value
	a1 = arg	Argument from Table E–11
Output:	v0 = 1	Success
	v0 = 0	Failure (not generated)
OpenVMS		
Input:	R16 = 6	Function code value
	R17 = arg	Argument from Table E–11
Output:	R0 = 1	Success
	R0 = 0	Failure (not generated)

Table E–4: 21164/21164PC Enable Counters

Bits	Meaning When Set
2	Operate on counter 2
1	Operate on counter 1
0	Operate on counter 0

Table E–5: 21164/21164PC Disable Counters

Bits	Meaning When Set
2	Operate on counter 2
1	Operate on counter 1
0	Operate on counter 0

Table E–6 21164 Select Desired Events

Bits	Name	Meaning
63:32		MBZ
31	PCSEL0	Counter 0 selection:
		Value Meaning
		0 Cycles
		1 Issues
30:25		MBZ
24:22	CBOX2	CBOX2 event selection (only has meaning when event selection field PCSEL2 is value <15>; otherwise MBZ). CBOX2 described in Table E–15.
21:19	CBOX1	CBOX1 event selection (only has meaning when event selection field PCSEL1 is value <15>; otherwise MBZ). CBOX1 described in Table E–14.
18:8		MBZ
7:4	PCSEL1	Counter 1 event selection. PCSEL1 described in Table E–12.
3:0	PCSEL2	Counter 2 event selection. PCSEL2 described in Table E–13.

Table E–7 21164PC Select Desired Events

Bits	Name	Meaning
63:32		MBZ
31	PCSEL0	Counter 0 selection:
		Value Meaning
		0 Cycles
		1 Issues
30:14		MBZ

Table E-7 21164PC Select Desired Events

Bits	Name	Meaning
13:11	PM1_MUX	PM1_MUX event selection (only has meaning when event selection field PCSEL2 is value <15>; otherwise MBZ). PM1_MUX is described in Table E-17.
10:8	PM0_MUX	PM0_MUX event selection (only has meaning when event selection field PCSEL1 is value <15>; otherwise MBZ). PM0_MUX is described in Table E-16.
7:4	PCSEL1	Counter 1 event selection. PCSEL1 described in Table E-12.
3:0	PCSEL2	Counter 2 event selection. PCSEL2 described in Table E-13.

Table E-8: 21164/21164PC Select Special Options

Bits	Meaning
63:31	MBZ
30	Stop count in user mode
29:10	MBZ
9	Stop count in PALmode
8	Stop count in kernel mode
7:1	MBZ
0	Monitor selected processes (when clear monitor all processes)

Setting any of the "NOT" bits causes the counters to not count when the processor is running in the specified mode. Under OpenVMS, "NOT_KERNEL" also stops the count in executive and supervisor mode, except as noted below:

NOT_BITS			Counters Operate Under These Modes When Bits Set:
K	U	P	
0	0	0	K E S U P
0	0	1	K E S U
0	1	0	K E S P
0	1	1	K E S
1	0	0	U P
1	0	1	U
1	1	0	P
1	1	1	E S (here "NOT_KERNEL" stops kernel counter only)

Note:

Tru64 UNIX and Alpha Linux counts user mode by using the executive counter; that is, the count for executive mode is returned as the user mode count.

Table E-9 contains the selection definitions for each of the three counters. All frequency fields are two-bit fields with the following values defined:

Table E-9: 21164/21164PC Select Desired Frequencies

Bits	Meaning When Set
63:10	MBZ
9:8	Counter 0 frequency:
	Value Meaning
	0 Do not interrupt
	1 Unused
	2 Low frequency ($2^{*}16$ (65536) events per interrupt)
	3 High frequency ($2^{*}8$ (256) events per interrupt)
7:6	Counter 1 frequency:
	Value Meaning
	0 Do not interrupt
	1 Unused
	2 Low frequency ($2^{*}16$ (65536) events per interrupt)
	3 High frequency ($2^{*}8$ (256) events per interrupt)
5:4	Counter 2 frequency:
	Value Meaning
	0 Do not interrupt
	1 Unused
	2 Low frequency ($2^{*}14$ (16384) events per interrupt)
	3 High frequency ($2^{*}8$ (256) events per interrupt)
3:0	MBZ

Table E-10: 21164/21164PC Read Counters

Bits	Meaning When Returned
63:48	Counter 0 returned value
47:32	Counter 1 returned value
31:30	MBZ
29:16	Counter 2 returned value
15:1	MBZ
0	Set means success; clear means failure

Table E–11: 21164/21164PC Write Counters

Bits	Meaning
63:48	Counter 0 written value
47:32	Counter 1 written value
31:30	MBZ
29:16	Counter 2 written value
15:0	MBZ

The values in Table E–12 choose the counter 1 (PCSEL1) event selection

Table E–12: 21164/21164PC Counter 1 (PCSEL1) Event Selection

Value	Meaning
0	Nothing issued, pipeline frozen
1	Some but not all issuable instructions issued
2	Nothing issued, pipeline dry
3	Replay traps (ldu, wb/maf, litmus test)
4	Single issue cycles
5	Dual issue cycles
6	Triple issue cycles
7	Quad issue cycles
8	Flow change (all branches, jsr-ret, hw_rei), where: If PCSEL2 has value 3, flow change is a conditional branch If PCSEL2 has value 2, flow change is a JSR-RET
9	Integer operate instructions
10	Floating point operate instructions
11	Load instructions
12	Store instructions
13	Instruction cache access
14	Data cache access
15	For the 21164, use CBOX1 event selection in Table E–14. For the 21164PC, use PM0_MUX event selection in Table E–16.

The values in Table E–13 choose the counter 2 (PCSEL2) event selection:

Table E–13: 21164/21164PC Counter 2 (PCSEL2) Event Selection

Value	Meaning
0	Long stalls (> 15 cycles)
1	Unused value
2	PC mispredicts
3	Branch mispredicts
4	I-cache misses
5	ITB misses
6	D-cache misses
7	DTB misses
8	Loads merged in MAF
9	LDU replays
10	WB/MAF full replays
11	Event from external pin
12	Cycles
13	Memory barrier instructions
14	LDx/L instructions
15	For the 21164, use CBOX2 event selection in Table E–15. For the 21164PC, use PM1_MUX event selection in Table E–17.

The values in Table E–14 choose the CBOX1 event selection.

Table E–14: 21164 CBOX1 Event Selection

Value	Meaning
0	S-cache access
1	S-cache read
2	S-cache write
3	S-cache victim
4	Unused value
5	B-cache hit
6	B-cache victim
7	System request

The values in Table E–15 choose the CBOX2 event selection.

Table E–15: 21164 CBOX2 Event Selection

Value	Meaning
0	S-cache misses
1	S-cache read misses
2	S-cache write misses
3	S-cache shared writes
4	S-cache writes
5	B-cache misses
6	System invalidates
7	System read requests

The values in Table E–16 choose the PM0_MUX event selection and perform the chosen operation in Counter 0.

Table E–16: 21164PC PM0_MUX Event Selection

Value	Meaning
0	B-cache read operations
1	B-cache D read hits
2	B-cache D read fills
3	B-cache write operations
4	Undefined
5	B-cache clean write hits
6	B-cache victims
7	Read miss 2 launched

The values in Table E–17 choose the PM1_MUX event selection and perform the chosen operation in Counter 1.

Table E–17: 21164PC PM1_MUX Event Selection

Value	Meaning
0	B-cache D read operations
1	B-cache read hits
2	B-cache read fills
3	B-cache write hits
4	B-cache write fills

Table E–17: 21164PC PM1_MUX Event Selection (Continued)

Value	Meaning
5	System read/flush B-cache hits
6	System read/flush B-cache misses
7	Read miss 3 launched

E.2.5 21264 and 21364 Performance Monitoring

PALcode instructions control the 21264 and 21364 onchip performance counters. For OpenVMS, the instruction is MTPR_PERFMON; for Tru64 UNIX and Alpha Linux, the instruction is wrperfmon.

The instruction arguments and results are described in the following sections. The scratch register usage is operating system specific.

Two 20-bit onchip counters count events. Counters can be individually programmed, read, and written.

Processes can be selectively monitored with the PME bit.

Supported counting modes differ between Pass 2.3 and Pass 3 (and subsequent) of the 21264. Pass 3 and subsequent passes, and the 21364 support the ProfileMe and Aggregate counting modes, while Pass 2.3 supports only Aggregate counting mode. When that distinction is relevant, it is documented. If no distinction is documented, the documentation applies to both Pass 2.3 and Pass 3 (and subsequent) of the 21264 and the 21364.

E.2.5.1 Performance Monitor Interrupt Mechanism

The performance monitoring interrupt mechanism varies according to the particular operating system.

For the OpenVMS Operating System

When a counter overflows and interrupt enabling conditions are correct, the counter causes an interrupt to PALcode. (For ProfileMe mode, the interrupt occurs after the ProfileMe window closes.) The PALcode builds an appropriate stack frame. The PALcode then dispatches in the form of an exception (not in the form of an interrupt) to the operating system by vectoring to the SCB performance monitor entry point through SCBB+650 (HWSCB\$Q_PERF_MONITOR), at IPL 29, in kernel mode.

An interrupt is generated for each counter overflow. For each interrupt, the status of each counter overflow is indicated by register R4:

R4 = 0 if performance counter 0 caused the interrupt
R4 = 1 if performance counter 1 caused the interrupt

When the interrupt is taken, the PC is saved on the stack frame as the old PC.

For the Tru64 UNIX and Alpha Linux Operating Systems

When a counter overflows and interrupt enabling conditions are correct, the counter causes an interrupt to PALcode. (For ProfileMe mode, the interrupt occurs after the ProfileMe window closes.) The PALcode builds an appropriate stack frame and dispatches to the operating system by vectoring to the interrupt entry point entINT, at IPL 6, in kernel mode.

An interrupt is generated for each counter overflow. For each interrupt, registers a0..a2 are as follows:

```

a0 = osfint$c_perf (4)
a1 = scb$v_perfmon (650)
a2 = 0 if performance counter 0 caused the interrupt
a2 = 1 if performance counter 1 caused the interrupt

```

E.2.5.2 Functions and Arguments

The functions execute only on a single (the current running) processor and are described in Table E–18.

The OpenVMS MTPR_PERFMON instruction is called with a function code in R16, a function-specific argument in R17, and any output is returned in R0.

The Tru64 UNIX and Alpha Linux wrperfmon instruction is called with a function code in a0, a function-specific argument in a1, and any output is returned in v0.

Table E–18 Performance Monitoring Functions

Function	Register Usage	Comments
Enable performance monitoring		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 1	Function code value
	a1 = arg	Argument from Table E–19
OpenVMS		
Input:	R16 = 1	Function code value
	R17 = arg	Argument from Table E–19
Disable performance monitoring		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 0	Function code value
	a1 = arg	Argument from Table E–20
OpenVMS		
Input:	R16 = 0	Function code value
	R17 = arg	Argument from Table E–20
Select desired events (MUX_SELECT)		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 2	Function code value
	a1 = arg	Argument from Table E–21
OpenVMS		
Input:	R16 = 2	Function code value
	R17 = arg	Argument from Table E–21

Table E–18 Performance Monitoring Functions (Continued)

Function	Register Usage	Comments
Select logging options		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 3	Function code value
	a1[0] = 1set = log all processes	
	a1[0] clear = log only selected processes	
OpenVMS		
Input:	R16 = 3	Function code value
	R17[0] set = log all processes	
	R17[0] clear = log only selected processes	
Read the counters		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 5	Function code value
Output:	v0 = contents of the counters; see Table E–22	
OpenVMS		
Input:	R16 = 5	Function code value
Output:	R0 = contents of the counters; see Table E–22	
Write the counters		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 6	Function code value
	a1 = arg	Argument from Table E–23
OpenVMS		
Input:	R16 = 6	Function code value
	R17 = arg	Argument from Table E–23
Enable and write selected counters		
Tru64 UNIX and Alpha Linux		
Input:	a0 = 7	Function code value
	a1 = arg	Argument from Table
OpenVMS		
Input:	R16 = 7	Function code value
	R17 = arg	Argument from Table

Table E–18 Performance Monitoring Functions (Continued)

Function	Register Usage	Comments
Read I_STAT values (ProfileMe mode only)		
Tru64 UNIX and Alpha Linux		
	Input: a0 = 8	Function code value
	Output: v0 = I_STAT values as shown in Table E–25	
OpenVMS		
	Input: R16 = 8	Function code value
	Output: R0 = I_STAT values as shown in Table E–25	
Read PMPC (ProfileMe mode only)		
Tru64 UNIX and Alpha Linux		
	Input: a0 = 9	Function code value
	Output: v0 = The PC of the last profiled instruction; see Table E–26	
OpenVMS		
	Input: R16 = 9	Function code value
	Output: R0 = The PC of the last profiled instruction; see Table E–26	

Table E–19 21264 and 21364 Enable Counters

R17/a1 Bits	Meaning When Set
1	Set I_CTL[PCT1_EN], which enables counter 1
0	Set I_CTL[PCT0_EN], which enables counter 0

Table E–20 21264 and 21364 Disable Counters

R17/a1 Bits	Meaning When Set
1	Clear I_CTL[PCT1_EN], which disables counter 1
0	Clear I_CTL[PCT0_EN], which disables counter 0

Table E–21 21264 and 21364 Select Desired Events

For 21264 to Pass 2.3¹:

R17/a1 Bits	Meaning	
4	Bit Value	Meaning
	1	Counter 0 counts retired instructions.
	0	Counter 0 counts cycles.
3–2	Bit Value	Meaning
	0000	Counter 1 counts cycles.
	0001	Counter 1 counts retired conditional branches.
	0010	Counter 1 counts retired branch mispredicts.
	0011	Counter 1 counts retired DTB single misses * 2.
	0100	Counter 1 counts retired DTB double double misses.
	0101	Counter 1 counts retired ITB misses.
	0110	Counter 1 counts retired unaligned traps.
	0111	Counter 1 counts replay traps.

¹ For 21264 Pass 3.0 and subsequent and 21364, see continuation of the table on the next page.

For 21264 Pass 3 and Subsequent (Including the 21364):

R17/a1 Bits	Meaning																				
4	<table border="0"> <thead> <tr> <th>Bit Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Enable ProfileMe mode.</td> </tr> <tr> <td>0</td> <td>Enable Aggregate mode.</td> </tr> </tbody> </table>	Bit Value	Meaning	1	Enable ProfileMe mode.	0	Enable Aggregate mode.														
Bit Value	Meaning																				
1	Enable ProfileMe mode.																				
0	Enable Aggregate mode.																				
3–2	<p>If bit 4 value is 1, enabling ProfileMe mode:</p> <table border="0"> <thead> <tr> <th>Bit Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Counter 0 counts retired instructions. Counter 1 counts cycles.</td> </tr> <tr> <td>01</td> <td>Counter 0 counts cycles. Counter 1 counts cycles of delayed retire pointer advance.</td> </tr> <tr> <td>10</td> <td>Counter 0 counts retired instructions. Counter 1 counts Bcache misses/long probe latency.</td> </tr> <tr> <td>11</td> <td>Counter 0 counts cycles.</td> </tr> </tbody> </table> <p>If bit 4 value is 0, enabling Aggregate mode:</p> <table border="0"> <thead> <tr> <th>Bit Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Counter 0 counts retired instructions. Counter 1 counts cycles.</td> </tr> <tr> <td>01</td> <td>Counter 0 counts cycles. Counter 1 is not defined.</td> </tr> <tr> <td>10</td> <td>Counter 0 counts retired instructions. Counter 1 counts Bcache misses/long probe latency.</td> </tr> <tr> <td>11</td> <td>Counter 0 counts cycles.</td> </tr> </tbody> </table>	Bit Value	Meaning	00	Counter 0 counts retired instructions. Counter 1 counts cycles.	01	Counter 0 counts cycles. Counter 1 counts cycles of delayed retire pointer advance.	10	Counter 0 counts retired instructions. Counter 1 counts Bcache misses/long probe latency.	11	Counter 0 counts cycles.	Bit Value	Meaning	00	Counter 0 counts retired instructions. Counter 1 counts cycles.	01	Counter 0 counts cycles. Counter 1 is not defined.	10	Counter 0 counts retired instructions. Counter 1 counts Bcache misses/long probe latency.	11	Counter 0 counts cycles.
Bit Value	Meaning																				
00	Counter 0 counts retired instructions. Counter 1 counts cycles.																				
01	Counter 0 counts cycles. Counter 1 counts cycles of delayed retire pointer advance.																				
10	Counter 0 counts retired instructions. Counter 1 counts Bcache misses/long probe latency.																				
11	Counter 0 counts cycles.																				
Bit Value	Meaning																				
00	Counter 0 counts retired instructions. Counter 1 counts cycles.																				
01	Counter 0 counts cycles. Counter 1 is not defined.																				
10	Counter 0 counts retired instructions. Counter 1 counts Bcache misses/long probe latency.																				
11	Counter 0 counts cycles.																				

Table E–22 21264 and 21364 Read Counters

R0/v0 Bits	Meaning When Returned
63–48	Reserved
47–28	Counter 0 returned value
27–26	Reserved
25–6	Counter 1 returned value
5–0	Reserved

Table E–23 21264 and 21364 Write Counters

R17/a1 Bits	Meaning
63–48	Reserved
47–28	Counter 0 value to write
27–26	Reserved
25–6	Counter 1 value to write
5–2	Reserved
1	When set, write to Counter 1
0	When set, write to Counter 0

Table E–24 21264 and 21364 Enable and Write Counters

R17/a1 Bits	Meaning
63–48	Reserved
47–28	Counter 0 value to write; writing zeroes clears the counter
27–26	Reserved
25–6	Counter 1 value to write; writing zeroes clears the counter
5–2	Reserved
1	When set, enable and write to Counter 1
0	When set, enable and write to Counter 0

Table E–25 21264 and 21364 Read I_STAT Values

R0/v0 Bits	Meaning
63–41	Reserved
40	ProfileMe mispredict trap If bit 39 is set, this bit indicates that the profiled instruction caused a mispredict trap. JSR/JMP/RET/COR or HW_JSR/HW_JMP/HW_RET/HW_COR mispredicts do not set this bit but can be recognized by the presence of one of these instructions at the PMPC location with bit 39 set. This identification is exact in all cases except error condition traps. Hardware corrected Icache parity or Dcache ECC errors, and machine check traps can occur on any instruction in the pipeline.
39	ProfileMe trap When set, indicates that the profiled instruction caused a trap. The trap type field, PMPC register, and instruction at the PMPC location are needed to distinguish all trap types.

Table E–25 21264 and 21364 Read I_STAT Values (Continued)

R0/v0 Bits	Meaning
38	<p>ProfileMe load-store order trap</p> <p>If the profiled instruction caused a replay trap, this bit set indicates that the precise trap cause was an Mbox load-store order replay trap. If clear, this bit indicates that the replay trap was any one of the following:</p> <ul style="list-style-type: none">• Mbox load-load order• Mbox load queue full• Mbox store queue full• Mbox wrong size trap (such as, STL → LDQ)• Mbox Bcache alias (2 physical addresses map to same Bcache line)• Mbox Dcache alias (2 physical addresses map to same Dcache line)• Icache parity error• Dcache ECC error

Table E–25 21264 and 21364 Read I_STAT Values (Continued)

R0/v0 Bits	Meaning																																		
37–34	ProfileMe trap types If the profiled instruction caused a trap (indicated by bit 39 set), this field indicates the trap type as follows:																																		
	<table border="1"> <thead> <tr> <th>Bit Value</th> <th>Trap Type</th> </tr> </thead> <tbody> <tr><td>15</td><td>Reset</td></tr> <tr><td>14</td><td>MT_FPCR</td></tr> <tr><td>13</td><td>Invalid (use PMPC, described below)</td></tr> <tr><td>12</td><td>Arithmetic</td></tr> <tr><td>11</td><td>Invalid (use PMPC, described below)</td></tr> <tr><td>10</td><td>Machine Check</td></tr> <tr><td>9</td><td>Invalid (use PMPC, described below)</td></tr> <tr><td>8</td><td>OPCDEC</td></tr> <tr><td>7</td><td>Dstream Fault</td></tr> <tr><td>6</td><td>DTB Single miss</td></tr> <tr><td>5</td><td>Unaligned Load/Store</td></tr> <tr><td>4</td><td>Floating point disabled</td></tr> <tr><td>3</td><td>DTB Double miss (4 level page tables)</td></tr> <tr><td>2</td><td>DTB Double miss (3 level page tables)</td></tr> <tr><td>1</td><td>Invalid (unused)</td></tr> <tr><td>0</td><td>Replay</td></tr> </tbody> </table>	Bit Value	Trap Type	15	Reset	14	MT_FPCR	13	Invalid (use PMPC, described below)	12	Arithmetic	11	Invalid (use PMPC, described below)	10	Machine Check	9	Invalid (use PMPC, described below)	8	OPCDEC	7	Dstream Fault	6	DTB Single miss	5	Unaligned Load/Store	4	Floating point disabled	3	DTB Double miss (4 level page tables)	2	DTB Double miss (3 level page tables)	1	Invalid (unused)	0	Replay
Bit Value	Trap Type																																		
15	Reset																																		
14	MT_FPCR																																		
13	Invalid (use PMPC, described below)																																		
12	Arithmetic																																		
11	Invalid (use PMPC, described below)																																		
10	Machine Check																																		
9	Invalid (use PMPC, described below)																																		
8	OPCDEC																																		
7	Dstream Fault																																		
6	DTB Single miss																																		
5	Unaligned Load/Store																																		
4	Floating point disabled																																		
3	DTB Double miss (4 level page tables)																																		
2	DTB Double miss (3 level page tables)																																		
1	Invalid (unused)																																		
0	Replay																																		

Traps due to ITB miss, Istream access violation, or interrupts are not reported in the trap type field because they do not cause pipeline aborts. Instead, those traps cause pipeline redirection and can be distinguished by examining the PMPC value for the presence of the corresponding PALcode entry offset addresses, shown below. In those cases, the ProfileMe interrupt is normally delivered when exiting the trap PALcode flow and the EXC_ADDR register contains the original PC that encountered the redirect trap.

PMPC[14:0]	Trap
0581	ITB miss
0481	Istream Access Violation
0681	Interrupt

Table E–25 21264 and 21364 Read I_STAT Values (Continued)

R0/v0 Bits	Meaning
33	ProfileMe Icache miss When set, indicates that the profiled instruction was contained in an aligned 4-instruction Icache fetch block that requested a new Icache fill stream.
32–30	ProfileMe counter 0 overcount When set, indicates a value (0-7) that must be subtracted from the counter 0 result to obtain an accurate count of the number of instructions retired in the interval beginning three cycles after the profiled instruction reaches pipeline stage 2 and ending four cycles after the profiled instruction is retired.
29–0	Reserved

Table E–26 21264 and 21364 Read PMPC Value

R0/v0 Bits	Meaning
63–2	Address of the profiled instruction
1	Reserved
0	When set, indicates that the PC field contains a physical-mode PALcode address.

Appendix F

Windows NT Software

This appendix contains a copy of the Windows NT part of the Rev 6 SRM and it is included for archival purposes only. There is no support for anything in this appendix.

This appendix describes how a particular implementation of the Windows NT Alpha operating system relates to the Alpha architecture. *It is important to note the following:*

- The interfaces described in this section will change as necessary to support the Microsoft Windows NT operating system.
- Effectively, many of the interfaces described in this section are private agreements between the PALcode and the kernel. Other software should not assume that those interfaces are available.
- In particular, the interfaces in this section must not be used by software developers who are writing device drivers; instead use the portable Windows NT device driver interfaces.
- The only interfaces in this section that may be used by nonsystem software are the bpt, rdteb, and gentrap PALcode instructions.

F.1 Introduction to Windows NT Alpha Software

The primary goal of the Windows NT Alpha PALcode implementation is total compatibility with the base operating system design and existing implementations of Windows NT for all processor architectures. *Maintaining compatibility with Windows NT and software portability between versions of Windows NT requires the stipulations mentioned in the introduction to this section. It is important that all software developers read those stipulations.*

The PALcode mechanism, coupled with the Windows NT Alpha design, provides binary compatibility for native system components across different processor implementations. The PALcode also provides a clean abstracted processor model that matches Windows NT requirements, requires minimal porting effort for new platforms, and provides the best possible performance while offering those features.

Windows NT Alpha is a 32-bit operating system. Therefore, the PALcode is a 32-bit implementation, with, for example, a 32-bit virtual address space. The internal processor registers are 32 bits, in canonical longword format. The page table entry (PTE) format is also 32 bits. The PALcode manages any required transformation between the 32-bit processor-independent formats and the 64-bit internal processor.

A Windows NT Alpha PALcode image is processor specific and platform independent. A single version of the PALcode (for a particular processor implementation) runs on all systems. The difference between processors is entirely hidden by the PALcode for each implementation. Thus, the PALcode interface allows the Windows NT Alpha operating system images to be binary-compatible across different processor implementations.

The PALcode image is read from the disk during the boot process, like all other components of the running operating system. The boot environment PALcode need only support the common swppal instruction to allow the operating system to load and initialize the PALcode.

Some functions and parameters must be implemented on a per-platform basis. Platform-dependent functions are implemented in the HAL (hardware abstraction layer), which is a system-specific library, loaded and dynamically linked at boot time.

The basic Windows NT Alpha design, therefore, consists of a platform-independent PALcode definition and binary-compatible kernel with system-dependent functions in the HAL.

The PALcode was designed to work smoothly and quickly with the Windows NT Alpha kernel. For example, the PALcode builds Windows NT Alpha trap frames and passes Windows NT Alpha status codes. Wherever possible, parameters and return values are passed in registers between the kernel and the PALcode.

The PALcode was also designed to keep dependencies on the kernel to a minimum. For example, only the processor control region and the kernel trap frame definition are shared between the PALcode and the Windows NT Alpha kernel.

F.1.1 Overview of System Components

The kernel is a binary-compatible image that can run on any Alpha processor, platform, or system. The kernel is binary compatible because of cooperation between it and other system components that provide the processor- and system-specific functions. Those cooperating components are the firmware, the OS Loader, the HAL (hardware abstraction layer), and the PALcode.

The firmware and OS Loader are the first components in the boot sequence and are responsible for establishing the environment in which the kernel, HAL, and PALcode execute. The kernel reads the configuration information provided by the firmware through the OS Loader and uses the standard interfaces provided by the HAL and the PALcode.

Firmware

The firmware contributes the following components to the boot sequence:

1. Establishes the privileged environment in which the OS Loader executes and the kernel begins executing (that is, provides memory management support and the swppal instruction).
2. Provides platform- and configuration-dependent services to the OS loader (such as I/O services) by using ARC call-back routines.
3. Creates the configuration database: devices, memory size, and so forth.
4. Reads the OS Loader from the disk and executes it.

OS Loader

The OS Loader is a linking loader that reads the component operating system images from the disk, performs necessary relocation, and binds the dynamically linked images together. The OS Loader loads the appropriate HAL and PALcode, based on the configuration information provided by the firmware.

The OS Loader loads the appropriate boot drivers as read from the operating system configuration files. The OS Loader also builds the loader parameter block structure by using information provided by the firmware. The loader parameter block includes configuration information (processor, system, device, and memory configuration) and per-processor data structures.

Once the operating system components are loaded, the OS Loader jumps to the beginning of the kernel to begin execution of the operating system. The OS Loader loads the operating system PALcode on a 64K-byte-aligned address. The kernel activates the operating system PALcode by executing the swppal instruction.

Hardware Abstraction Layer (HAL)

The HAL provides the system-specific layer between the kernel and the system hardware. The HAL provides interfaces for the following types of functions:

- Interrupt handling, including dispatch and acknowledge
- DMA control
- Timer support
- Low-level I/O support
- Cache coherency

If a processor implementation requires PALcode intervention to support any of those functions, then the PALcode must support those processor-specific functions in a system-independent manner.

PALcode

The PALcode is specific to a particular processor implementation and must hide the internal workings of the processor from the kernel. The PALcode for a particular processor may include per-processor functions, but they must be called only by the HAL.

F.1.2 Calling Standard Register Usage

Table F–1 General-Purpose Integer Registers

Register Number	Symbolic Name	Volatility	Description
r0	v0	Volatile	Return value register
r1 – r8	t0 – t7	Volatile	Temporary registers
r9 – r14	s0 – s5	Nonvolatile	Saved registers
r15	s6/fp	Nonvolatile	Saved register/frame pointer
r16 – r21	a0 – a5	Volatile	Argument registers
r22 – r25	t8 – t11	Volatile	Temporary registers
r26	ra	Volatile	Return address register
r27	t12	Volatile	Temporary register
r28	at	Volatile	Assembler temporary register
r29	gp	Nonvolatile	Global pointer
r30	sp	Nonvolatile	Stack pointer
r31	zero	Constant	RAZ / writes ignored

Table F–2 General-Purpose Floating-Point Registers

Register Number	Volatility	Description
f0	Volatile	Return value register (real part)
f1	Volatile	Return value register (imaginary part)
f2 – f9	Nonvolatile	Saved registers
f10 – f15	Volatile	Temporary registers
f16 – f21	Volatile	Argument registers
f22 – f30	Volatile	Temporary registers
f31	Constant	RAZ / writes ignored

F.1.3 Code Flow Conventions

The code flows are shown as an ordered sequence of instructions. The instructions in the sequence may be reordered as long as the results of the sequence of instructions are not altered. In particular, if an instruction j is listed subsequent to an instruction i and i writes any data that is used by j , then i must be executed before j .

F.2 Processor, Process, Threads, and Registers

This section describes structures and registers that support the processor, process, and thread environment.

F.2.1 Processor Status

The processor status register (PSR) defines the processor status. The PSR is shown in Figure F-1 and described in Tables F-3, F-4, and F-5.

Figure F-1: Processor Status Register



Table F-3 Processor Status Register Fields

Field	Type	Description
IRQL	RW	Interrupt request level, in the range 0–7, as described in Table F-4. Any interrupt disabled at a lower priority level is also disabled at a higher priority level.
IE	RW	Interrupt enable: 0 = interrupts disabled 1 = interrupts enabled A global interrupt enable to turn interrupts on and off without changing the IRQL.
MODE	RW	Processor mode: 0 = kernel mode 1 = user mode Describes the current processor privilege mode: user (unprivileged) or kernel (privileged). The processor privilege mode defines the instructions that can be executed and the memory protection that is used, as described in Table F-5.

Table F-4 Processor Status Register IRQL Field Summary

IRQL	Name	Description
0	PASSIVE_LEVEL	All interrupts enabled.
1	APC_LEVEL	APC software interrupts disabled.
2	DISPATCH_LEVEL	Dispatch software interrupts disabled.
3	DEVICE_LEVEL	Low-priority device hardware interrupts disabled.
4	DEVICE_HIGH_LEVEL	High-priority device hardware interrupts disabled.
5	CLOCK_LEVEL	Clock hardware interrupts disabled.
6	IPI_LEVEL	Interprocessor hardware interrupts disabled.
7	HIGH_LEVEL	All maskable interrupts disabled.

Table F–5 Processor Privilege Mode Map

Operation	Privileged	Unprivileged
Superpage access	Yes	No
Page protection	Access to all pages	Access to only those pages with the Owner bit = 1
Privileged PALcode instructions	Yes	No

F.2.2 Internal Processor Register Summary

The internal processor registers in Table F–6 are defined across all implementations. Implementation of these registers within the processor is implementation dependent.

Table F–6 Internal Processor Register Summary

Name	Initial Value	Description
ASN	0	Address space number of owning process of current thread
GENERAL_ENTRY	0	General exception class kernel handler address
IKSP	0	Initial kernel stack pointer
INTERRUPT_ENTRY	0	Interrupt exception class kernel handler address
KGP	0	Kernel global pointer
MCES	1	Machine check error summary
MEM_MGMT_ENTRY	0	Memory management exception class kernel handler address
PAL_BASE	1	PALcode image base address
PANIC_ENTRY	0	Panic exception class kernel handler address
PCR	1	Processor control region base address
PDR	0	Page directory base address
PSR	1	Processor status register
RESTART_ADDRESS	1	Restart execution address
SIRR	0	Software interrupt request register
SYSCALL_ENTRY	0	System service exception class kernel handler address
TEB	0	Thread environment block base address
THREAD	0	Thread unique value (kernel thread address)

¹ The register has an architected initial value. See the register description in Table F–7.

F.2.3 Internal Processor Registers

Table F-7 lists and describes the internal processor registers.

Table F-7 Internal Processor Registers

Name	Description
ASN	<p>Address space number of owning process of current thread</p> <p>Bits <15:0> of the ASN register contain the address space number for the current process. Bits <31:16> are RAZ.</p> <p>The ASN is a process tag that is used by the processor to qualify each virtual translation. When translations are qualified, it is not necessary for the processor to flush all virtual translations for previous processes when performing a context swap or process swap. The swpctx and swpprocess instructions provide the ASN.</p>
GENERAL_ENTRY	<p>General exception class kernel handler address</p> <p>The GENERAL_ENTRY register contains the entry address (in 32-bit superpage format) for the kernel exception handler for the General class of exceptions. The wrentry instruction writes GENERAL_ENTRY.</p>
IKSP	<p>Initial kernel stack pointer</p> <p>The IKSP register contains the initial kernel stack address. IKSP points to the top of the kernel stack for the currently executing thread. The rdksp instruction reads IKSP and the swpksp instruction writes IKSP. IKSP is also written by swpctx and during system initialization by initpal.</p>
INTERRUPT_ENTRY	<p>Interrupt exception class kernel handler address</p> <p>The INTERRUPT_ENTRY register contains the entry address (in 32-bit superpage format) of the kernel exception handler for the Interrupt class of exceptions. The wrentry instruction writes INTERRUPT_ENTRY.</p>
KGP	<p>Kernel global pointer</p> <p>The KGP register contains the kernel global pointer, the gp value. The PALcode restores the kernel global pointer to the general-purpose register gp whenever dispatching to a kernel exception handler. The initpal instruction writes the KGP.</p>
MCES	<p>Machine check error summary</p> <p>The MCES register is used to report and control the current state of machine check handling. The MCES register contains multiple fields that are described in Section F.4.3. The initial values for the MCES register fields DSC, DPC, and DMK are implementation specific, and all other fields set to 0. The recommended initial values are DMK = 0, DPC = 1, and DSC = 1.</p>

Table F–7 Internal Processor Registers (Continued)

Name	Description
MEM_MGMT_ENTRY	<p>Memory management exception class</p> <p>The MEM_MGMT_ENTRY register contains the entry address (in 32-bit superpage format) of the kernel exception handler for the Memory Management class of exceptions. The wrentry instruction writes MEM_MGMT_ENTRY.</p>
PAL_BASE	<p>PALcode image base address</p> <p>The PAL_BASE register contains the physical address of the base of the currently active PALcode image. Its initial value is the address of the PALcode entry point. PAL_BASE controls which PALcode image is currently active and is written during PALcode initialization. The PAL_BASE register is illustrated and described in Section F.6.2.</p>
PANIC_ENTRY	<p>Panic exception class kernel handler address</p> <p>The PANIC_ENTRY register contains the entry address (in 32-bit superpage format) of the kernel exception handler for the Panic class of exceptions. The wrentry instruction writes PANIC_ENTRY.</p>
PCR	<p>Processor control region base address</p> <p>The PCR register contains the base address (in 32-bit superpage format) of the processor control region page. The processor control region is a page of per-processor data. The PCR is passed as an initialization parameter and the rdpcr instruction reads it.</p>
PDR	<p>Page directory base address</p> <p>The PDR register contains the base physical address of the page directory page. The page directory page contains all of the first-level page table entries (the page directory entries or PDEs). As such, the page directory page defines an address space for a process. The swpctx and swpprocess instructions write the PDR when the address space is swapped. The initpal instruction also writes the PDR.</p>
PSR	<p>Processor status register</p> <p>The PSR controls the privilege state and interrupt priority of the processor. The PSR register contains multiple fields that are described in Section F.2.1. The initial values for the fields in the PSR are IRQL=7, IE=1, and MODE=0 (kernel).</p>
RESTART_ADDRESS	<p>Restart execution address</p> <p>The RESTART_ADDRESS register contains the address where the processor resumes execution when the PALcode exits. For example, upon entry to each of the PALcode instructions, the RESTART_ADDRESS register contains the <i>virtual address</i> + 4 of that instruction. The initial value of the RESTART_ADDRESS register is the kernel initialization continuation address, passed as a parameter to the initialization routine.</p>

Table F–7 Internal Processor Registers (Continued)

Name	Description
SIRR	Software interrupt request register The SIRR register indicates requested software interrupts. SIRR contains multiple fields that are defined in Section F.4.2.7.
SYSCALL_ENTRY	System service exception class kernel handler address The SYSCALL_ENTRY register contains the entry address (in 32-bit superpage format) of the kernel exception handler for the System Service class of exceptions. The wrentry instruction writes SYSCALL_ENTRY.
TEB	Thread environment block base address The TEB register contains the address of the user thread environment block. Each swpctx instruction writes the TEB; the rdteb instruction reads it.
THREAD	Thread unique value (kernel thread address) The THREAD register contains the address of the currently executing kernel thread structure. Each swpctx instruction writes the THREAD register; the rdthread instruction reads it.

F.2.4 Processor Data Areas

The operating system per-processor data structure is the processor control region. The processor control region is a one-page (superpage) data structure that stores information that may be specific to a particular architecture. This information is data that is shared between the PALcode, the HAL, and/or the architecture-specific portions of the kernel. See Section F.3.1 for information on the superpage.

F.2.4.1 Processor Control Region

The processor control region contains a number of data structures that are of importance to the PALcode, including:

- A 3064-byte region that is reserved for the PALcode and is the only per-processor data region available to the PALcode.
- The interrupt level table (ILT), which maps the interrupt enable masks for each possible interrupt request level. The PALcode may continually read these masks or may read them once and cache them inside the processor.
- The interrupt dispatch table (IDT), which contains the address of an interrupt handler for each possible interrupt vector.
- The interrupt mask table (IMT), which maps each possible pattern of interrupt requests to the highest priority interrupt vector and the corresponding synchronization level.
- The panic stack pointer.
- The restart block pointer.
- The firmware restart address.

The PALcode is responsible for initializing the PALcode base address field and several PALcode revision fields within the processor control region.

The `rdpcr` instruction returns the base address of the processor control region.

F.2.4.2 PALcode Version Control

The PALcode is responsible for writing version information in the processor control region. The `PalMajorVersion`, `PalMinorVersion`, and `PalSequenceVersion` are provided for maintenance and debugging. The PALcode writes these fields, but the values are implementation specific.

The kernel may use the `PalMajorSpecification` and `PalMinorSpecification` fields for check-pointing with the PALcode.

The PALcode writes the specification fields with version numbers that correspond to the version of the specification to which the PALcode image complies. Minor revisions within the same major revision are backwards compatible. The kernel may read the `PalMajorSpecification` and determine if it is compatible with the version of the PALcode. If the kernel is not compatible (if the `PalMajorSpecification` is greater than the kernel's expected PALcode major specification), the kernel runs down in a controlled manner.

The version agreement between the PALcode and the kernel is a private agreement between these two system components. No other system component, including the HAL and device drivers, may depend on any values from those fields.

F.2.4.3 PALcode Alignment Fixup Count

PALcode must maintain a count in the processor control region `PalAlignmentFixupCount` field of the total number of alignment fixups that the PALcode accomplishes. `PalAlignmentFixupCount` is an unsigned quadword field that is incremented by one when the PALcode fixes up an alignment fault. The field silently overflows to zero.

The kernel may use the `PalAlignmentFixupCount` field for determining the total number of alignment fixups on a system by adding the value in that field for each processor to the number of alignment fixups done by the kernel.

F.2.5 Caches and Cache Coherency

Implementations may include caches that are not kept coherent with main memory. The `imb` instruction provides an architected common way to make the instruction execution stream coherent with main memory. The `imb` instruction guarantees that subsequently executed instructions are fetched coherently with respect to main memory on only the current processor.

User-mode code that directly modifies the instruction stream, either through writes or by DMA from an I/O device, must call the appropriate Windows NT API to ensure I-cache coherency. User-mode code that uses standard APIs to modify the instruction stream works as expected and is handled by the APIs themselves.

F.2.6 Stacks

There are four stacks:

- Kernel stack

Each thread is allocated its own pages for a kernel stack. The kernel stack is the two pages of virtual address space below the IKSP for a thread, where the IKSP points to the byte beyond the top of the two pages. The initial kernel stack pointer (IKSP) points to the top of the currently active kernel stack for the current thread. Two PALcode instructions provide access to the IKSP: rdksp to read the IKSP and swpksp to atomically read the current IKSP and write a new one.

Must remain valid for the currently executing thread. Software must guarantee that the kernel stack pointer remains 16-byte aligned.

- User Stack

A per-thread stack on which all user-mode components are executed.

- Deferred procedure call (DPC) stack

A processor-wide stack upon which all deferred procedure calls are executed. Must remain valid for the lifetime of the system.

- Panic stack

Allows the operating system to remain coherent through a system crash. Must remain valid for the lifetime of the system.

The kernel, DPC, and panic stacks execute in kernel mode; the user stack executes in user mode.

F.2.7 Processes and Threads

Windows NT Alpha is designed as a multithread operating system with multiple threads executing within the same process. Each thread has its own processor context, user-mode stack, and kernel stack. Memory and the address space are shared across all threads in the same process.

The PALcode "knows" nothing about the structure of threads or processes. The PALcode implements the means to swap from one thread context to another and to allow a thread to attach to the address space of another process.

The state to accomplish these operations is passed entirely in registers. The PALcode maintains the THREAD and TEB internal processor registers. They allow threads to query about the state of the currently executing thread.

The THREAD register, a unique value identifying the current thread, is written when the thread context is swapped. The privileged instruction rdthread reads the THREAD register.

The TEB register, a user-accessible pointer to the thread environment block for the new thread, is written when thread context is swapped. The unprivileged rdteb instruction reads the TEB register. Again, the PALcode knows nothing about the structure of the thread environment block; the PALcode simply maintains the TEB register value when context is switched.

F.2.7.1 Swapping Thread Context to Another Thread

The swpctx instruction swaps the context from one thread to another thread. The following parameters are passed to swpctx:

Initial kernel stack pointer

Swpctx must switch to the new kernel stack for the new thread. The initial kernel stack pointer is written to the internal processor register IKSP.

THREAD internal processor register (unique thread value)

TEB internal processor register (thread environment block pointer)

These registers are maintained by the kernel and only written during a context switch. Implicitly, the values in these registers for a particular thread cannot change while that thread is executing.

PFN of the directory table base page for the new process

ASN for the new process

ASN_wrap_indicator

The PFN and ASN allow switching to a new process address space. The PFN of the directory table base page is an overloaded parameter; it is used to indicate if the process needs to be swapped.

- The PFN is set to a negative value in the kernel if the previous thread and the new thread are in the same process (address space). There is no need to swap the address space if the two threads are in the same process. The values for the ASN parameters are then UNPREDICTABLE.
- If the two threads are in different processes, the PFN is greater than or equal to zero and is used to write the PDR internal processor register. When the PFN is valid (greater than zero), the ASN must also be valid and is used to write the ASN internal processor register.

Swapping to a new process address space involves establishing a new directory pointer to the page table base page for the new process and possibly performing translation buffer operations. A set ASN_wrap_indicator signals that the PALcode must perform an invalidation operation for each cached translation in the translation buffers and virtual caches that does not have the address space match (ASM) bit set.

F.2.7.2 Swapping Thread Context to Another Process

The swpprocess (swap process) instruction allows a thread to attach to another process (in another address space). Swpprocess requires the PFN of the new directory table base page and the new ASN as input. Swpprocess performs the same address space swapping operation as does swpctx when the PFN of the page directory is valid.

F.3 Memory Management

F.3.1 Virtual Address Space

Windows NT Alpha is a 32-bit implementation with a 32-bit virtual address space, as represented in Table F–8.

Table F–8 Virtual Address Map

Address Range ₁₆ (32 bits)	Permission	Description
00000000–7FFFFFFF	User and Kernel	General user address space
80000000–BFFFFFFF	Kernel	Nonmapped kernel space (32-bit superpage)
C0000000–C1FFFFFF	Kernel	Mapped, page table space
C2000000–FFFFFFF	Kernel	Mapped, general kernel space

The address map takes advantage of the 32-bit superpage feature of the Alpha architecture. If the implementation of the 32-bit superpage is not done in hardware, it must be implemented in software (PALcode). The entire 1-GB address space mapped by the 32-bit superpage must be valid at all times for both instruction fetch and data access.

Implementation Note (Hardware):

It is strongly recommended that implementations include a hardware mapping of the 32-bit superpage for both instruction and data stream.

F.3.2 I/O Space Address Extension

The Windows NT Alpha kernel implementation takes advantage of the architecture's 64-bit address space to provide a nonmapped extended address for I/O space. The extended address space uses the 43-bit superpage that is available in the Alpha architecture. The superpage allows kernel mode access to an address space with a predetermined translation. Therefore, those accesses never require page table mapping or cause a translation buffer miss.

Implementation Note:

The extended address space is particularly important to Alpha implementations that do not include the BWX extension, because the bus mapping scheme for those implementations uses a shifted physical address, where the lower address bits are used to determine the byte enables. Therefore, the effective page size is smaller. See Appendix D for information about the BWX extension.

The extended superpage provides nonmapped access to a 41-bit physical address space. The nonmapped superpage I/O accesses provide Alpha systems with a performance advantage because there is no need to write as many page table entries and to fill as many translation buffer misses as would be necessary without it. The extended address space is desirable because the likely physical address space is 34 bits or more and the 32-bit superpage can only allow accesses to 30 bits of physical address space. The extended address space is the only

exception to the 32-bit virtual address map shown in Table F–8. The extended address space is intended for I/O access only and can only be used in kernel mode. The address mapping for the extended address space is shown in Table F–9.

Table F–9 I/O Address Extension Address Map

Address Range ₁₆ (64 bits)	Permission	Description
FFFFFC0000000000– FFFFFFFD- FFFFFFFF	Kernel	Nonmapped kernel mode I/O extension

F.3.3 Canonical Virtual Address Format

All virtual addresses, with the exception of the large superpage addresses, must be in canonical longword form. The PALcode must check the faulting virtual addresses in the first level miss flows and raise an exception if the addresses are not canonical longwords. The check is required because the processor may generate 64-bit addresses that are not canonical longwords, but the common memory management code only knows about 32-bit addresses and so cannot necessarily identify or signal the exception to the offending code. The PALcode cannot simply resolve the miss by using only the lower 32 bits. When the faulting instruction is re-executed, it attempts again to access the noncanonical address. If a virtual address fails the canonical form test, the PALcode raises a general exception (see Section F.4.1.7).

F.3.4 Page Table Entries

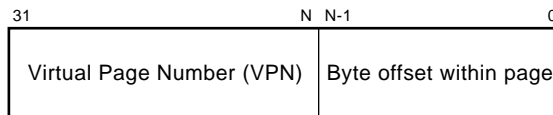
Page table entries (PTEs) provide the translation from virtual addresses to their physical addresses. The PTE includes the physical address in the form of a page frame number (PFN), protection information, and performance hints. The virtual address is related to a page table entry based solely upon the position of the PTE within a set of page tables.

Two methods may be used to traverse the page tables to retrieve the corresponding PTE for a given virtual address. The first is to view the page tables as a single-level virtually contiguous table. The second is to view the page tables as a two-level physical table.

F.3.4.1 Single-Level Virtual Traversal of the Page Tables

For a single-level virtual traversal, a virtual address must be viewed as shown in Figure F–2, where $2^{*}N$ is the implementation page size.

Figure F–2: Virtual Address (Virtual View)



To access the corresponding PTE for a VA (virtual address) using the single-level virtual method, use the following algorithm.

```
! In the algorithm:
!   VIRTUAL_PTE_BASE = C000000016
!   PAGE_SHIFT = N
! Clear upper bits in case va is sign-extended:
```



```

va ← BYTE_ZAP( va, F0 )
! Get virtual page number:
vpn ← RIGHT_SHIFT( va, PAGE_SHIFT )
! 4 bytes per pte, offset + base:
pte_va ← VIRTUAL_PTE_BASE + ( vpn * 4 )
! Do a virtual load of pte:
pte ← (pte_va)

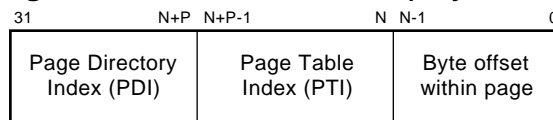
```

F.3.4.2 Two-Level Physical Traversal of the Page Tables

The two-level physical method can be used to find the corresponding PTE for a virtual address when the virtual access method cannot be used (for example, if the PTE address is not valid). The key to physically traversing the page tables is the PDR internal processor register. The PDR is maintained on a per-process basis whenever process context is swapped. The PDR is the physical address of the page directory page that forms the first level of the page tables. The first level of the page tables easily fits within a single page. Each entry in the page directory page is called a PDE (page directory entry). One PDE maps one page of PTEs.

A virtual address must be viewed as shown in Figure F-3 for a two-level, physical traversal of the page tables. In Figure F-3, $2^{*}N$ is the implementation page size, and $2^{*}P$ is (PTEs per page = page size / 4).

Figure F-3: Virtual Address (Physical View)



The following algorithm uses the two-level physical traversal method to access the corresponding PTE for a VA (virtual address).

```

! In the algorithm:
!   PDE_SHIFT = N + P
!   PAGE_SHIFT = N

! Clear upper bits in case va is sign-extended:
va ← BYTE_ZAP( va, F0 )
! Get pde number:
pde_index ← RIGHT_SHIFT( va, PDE_SHIFT )
! 4 bytes per pde, index * 4 byte offset:
pde_offset ← pde_index * 4
! Offset + base:
pde_pa ← PDR + pde_offset
! Do a physical load of the page directory entry:
pde ← (pde_pa)
! Get PFN of pte page from pde:
pte_pfn ← pde<PFN>
! Get physical address of pte page:
pte_page ← LEFT_SHIFT( pte_pfn, PAGE_SHIFT )
! Extract page table index from virtual address:
pte_index ← va<pti>
! Calculate offset, 4 bytes per pte:
pte_offset ← pte_index * 4

```

```

! Address base + offset:
pte_pa ← pte_page + pte_offset
! Do a physical load to read the pte:
pte ← (pte_pa)

```

Page directory entries are themselves page table entries and so they have the same format. There are some implications for DTB implementation because the PDEs establish a recursive mapping for addresses within the PTE address space. The implications and a description of the recursive mapping are described in Section F.3.6.

F.3.4.3 Page Table Entry Summary

The format for a PTE is shown in Figure F–4 and described in Table F–10.

Figure F–4: Page Table Entry

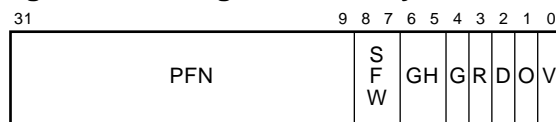


Table F–10 Page Table Entry Fields

Field	Description
PFN	Page frame number
SF	Reserved for software (operating system)
GH	Granularity hints Optional hint that provides for mapping translations larger than the standard implementation page size. These large pages must be both virtually and physically aligned. Defines the translation in terms of a multiple of the page size, where the multiplier equals $8 \times N$, where N is the granularity hint value in the range 0–3.
G	Global translation hint (address space match) Optional hint that the indicated translation is global for all processes.
R	Reserved

Table F–10 Page Table Entry Fields (Continued)

Field	Description
D	Dirty: 0 = page is not dirty 1 = page is dirty Implemented as the inverse of fault on write (FOW). Serves double duty by causing faults for the first write to a page. Serves as a write-protect bit and as a marker that allows the operating system to track dirty pages.
O	Owner: 0 = kernel access only 1 = user access permitted Indicates whether user mode is allowed across this page, either for instruction fetch or data access. Kernel mode code has implied access to all pages that have a valid translation.
V	Valid: 0 = translation not valid 1 = valid translation

F.3.5 Translation Buffer Management

As shown in Table F–11, the PALcode provides the `dtbis`, `tbia`, `tbim`, `tbimasn`, `tbis`, and `tbisasn` instructions to manage the cached virtual translations maintained in the translation buffers and virtual caches.

Table F–11 Translation Buffer Management Instructions

Instruction	Operation
<code>dtbis</code>	Invalidates a single data stream translation for a specified address. It is designed for those cases when the operating system can determine that the translation is not used in the instruction stream. Implementations may advantageously use <code>dtbis</code> to avoid needing to invalidate instruction stream translations in both an instruction TB and a virtual I-cache.
<code>tbia</code>	Invalidates all page table translations for both instruction and data stream access. The translations invalidated are limited to "page table translations" because it is possible that an implementation has used fixed TB entries to implement one or more of the required superpages. These fixed translations are considered "hard-wired" by the operating system and must be valid at all times.
<code>tbim</code>	Invalidates multiple virtual translations, passed as a parameter, for the current ASN. <code>Tbim</code> invalidates translations for both instruction and data stream access.

Table F–11 Translation Buffer Management Instructions (Continued)

Instruction	Operation
tbimasn	Invalidates multiple virtual translations for a specified address space number (ASN), passed as a parameter. The ASN may or may not be the currently executing thread. Tbimasn invalidates translations for both instruction and data stream access.
tbis	Invalidates a single translation for a specific virtual address, passed as a parameter. Tbis invalidates the translation for both instruction and data stream access.
tbisasn	Invalidates a translation for a single virtual address for a specified address space number (ASN). The ASN may or may not be for the currently executing thread. Tbisasn invalidates the translation for both instruction and data stream access.

On processors that implement physical, noncoherent instruction caches, instructions that invalidate I-stream translations must also invalidate instruction cache blocks from the physical pages that correspond to the invalidated virtual translations.

F.3.6 Implications of Recursive TB Mapping

Recursive virtual mapping has an implication for data translation buffer implementations: it is possible for two identical translations to be written in the DTB during the same miss handling sequence. If the DTB cannot correctly operate with two identical translations, the PALcode must include additional checks to prevent the condition from occurring.

The page tables can be viewed either as a virtual contiguous single-level table or as a two-level table that must be traversed physically. When viewed as a two-level table, the first level is a single page called the page directory page. Each page directory page entry, called a PDE, provides the first-level translation so that the TB-fill code can find the page table page that contains the PTE with the translation for the faulted virtual address. All page table pages are mapped by a PDE in the page directory page.

The page tables are recursive. The page directory page is a standard page table page and it is virtually mapped in the single-level virtual page table. Therefore, there exists one PDE that maps the page directory page. The PDE that maps the page directory page in a two-level lookup is also the PTE that maps the page directory page for the single-level virtual mapping. This special PDE is called the root PTE or RPTE.

Assume that the processor implementation has two data stream TB miss flows — one for the misses taken in native mode and one for the misses taken in the PALcode environment. For the case when a native-mode virtual access is made to the page directory page, PALcode takes the following flows:

Because there is only one PTE, RPTE, that exhibits this behavior, the PALcode can check the faulting PTE address in the second-level fill routine to special case for RPTE. It is preferable not to slow down even the second-level fill flow. However, this is a processor implementation decision

Native Miss Flow	PALcode Environment Miss Flow
<ol style="list-style-type: none"> 1. {get va for PTE that maps the faulted va: VA} 2. {get the PTE using its va} ldl rx, 0(ry) where ry ← va of PTE 7. load completes, rx ← RPTE} 8. {write the translation for the faulting va, VA, into the DTB} 9. { RPTE is now in the DTB twice} 10. {Re-execute the original native-mode instruction that faulted when accessing VA} 	<ol style="list-style-type: none"> 3. {ldl rx, 0(ry) from PALcode environment faulted} 4. {resolve this fault by making the va of the missed PTE valid} 5. {translation for RPTE is written into the DTB} 6. {re-execute the load that failed since the va of the PTE is now valid}

F.4 Exceptions, Interrupts, and Machine Checks

At certain times during the operation of a system, events within the system require the execution of software outside the explicit flow of control. When such an exceptional event occurs, an Alpha processor forces a change in control flow from that indicated by the current instruction stream. The notification process for such events is an exception, an interrupt, or a machine check.

F.4.1 Exceptions

F.4.1.1 Exception Dispatch

When the processor encounters an exception, it traps to PALcode that provides preliminary exception dispatch for the operating system. Some exceptions, such as TB miss, may be handled entirely by the PALcode without the intervention of the operating system.

The PALcode provides a simple and efficient method of dispatching to the operating system for those exceptions that require operating system action. In general, the following operations characterize exception dispatch:

1. Switch to kernel mode (if in user mode).
2. Allocate a trap frame on the kernel stack.
3. Save the necessary processor state in the trap frame.
4. Prepare arguments to the kernel exception handler using the standard argument registers where possible.
5. Set the processor state for executing the kernel (establish the stack pointer so it points to the kernel stack, establish the global pointer to point to the kernel global area).
6. Restart execution at the address of the kernel exception handler registered for the class of exception that was encountered.

F.4.1.2 Exception Classes

The PALcode classifies each exception into one of the following categories:

- Memory management exceptions

Memory management exceptions, described in Section F.4.1.5, are raised for:

- Translation not valid faults: accesses to addresses that do not have a valid translation for the currently executing context
- Access violations: accesses to addresses for which the currently executing context does not have permission for the access

- System service call exceptions

Although not really exceptions, system service calls are handled as exceptions to allow unprivileged code to request and receive privileged services. System services may be requested from both unprivileged and privileged modes (user and kernel mode respectively). System service calls are described in Section F.4.1.6.

- General exceptions

The general exception class, described in Section F.4.1.7, is the catchall category for all of the other exceptions that may be raised by unprivileged code:

- Arithmetic exceptions
- Unaligned memory access
- Illegal instruction execution
- Invalid (non-canonical virtual) address exceptions

- Software exceptions
- Breakpoints
- Subsetted instruction execution
- Panic exceptions
 - The panic exception class, described in Section F.4.1.8, is reserved for conditions from which execution cannot reliably be continued. The following general cases of panic exceptions are anticipated:
 - Invalid kernel stack (including overflow and underflow)
 - Unexpected exceptions from PALcode

F.4.1.3 Returning from Exceptions

The rfe and retsys instructions are provided for returning from exceptions.

The rfe (return from exception or interrupt) instruction allows the operating system to return from an exception. Rfe may also be used to transition from kernel mode to user-mode startup code.

The rfe instruction reverses the effect of an exception by restoring the original processor state from the trap frame on the kernel stack. In addition, rfe accepts a parameter that allows it to set software interrupt requests for the execution context that is about to be reestablished

Two exception classes do not use rfe to return to the previously executing context: system service call and panic exceptions. The retsys instruction is used for returning from system service call exceptions because a system service call has different semantics with regard to the saved processor state than the other exceptions.

Panic exceptions do not return because they precipitate a controlled crash of the operating system.

F.4.1.4 Trap Frames

Trap frames are allocated on the kernel stack for all classes of exceptions in PALcode. The PALcode also partially writes the trap frame; the fields written are based upon the exception being handled. The kernel stack must be guaranteed to remain aligned on a 16-byte boundary, as specified in the Windows NT Alpha calling standard. The trap frame itself is guaranteed in size to be a multiple of 32 bytes. The PALcode may over-align the kernel stack pointer when allocating the trap frame in order to improve memory throughput, with consideration for the extra memory being consumed. The trap frame is structured so that writes aggregate. The register values stored in the trap frame are 64-bit values. This is required as the register set is 64 bits and may contain 64-bit values (as opposed to canonical longwords).

Trap frame definitions are shown in Table F–12.

Table F–12 Trap Frame Definitions

Symbolic Name	Size	Description
TrIntSp	Quadword	Stack pointer register at point of exception
TrPsr	Longword	Processor status register at point of exception
TrFir	Quadword	Exception program counter
TrIntA0	Quadword	Register a0 at point of exception
TrIntA1	Quadword	Register a1 at point of exception
TrIntA2	Quadword	Register a2 at point of exception
TrIntA3	Quadword	Register a3 at point of exception
TrIntFp	Quadword	Frame pointer register at point of exception
TrIntGp	Quadword	Global pointer register at point of exception
TrIntRa	Quadword	Return address register at point of exception

F.4.1.5 Memory Management Exceptions

PALcode recognizes two classes of memory management exceptions: translation not valid faults and access violations. Translation not valid faults are detected when a page table entry for a virtual address has the valid bit cleared. The invalid page table entry can be either a first- or second-level table entry. Access violations are detected by the hardware when the processor attempts to access a virtual address and that type of access is not permitted according to the protection mask in the page table entry that maps the translation for the virtual address.

The PALcode dispatches to the kernel in the same manner for each of these two classes of exceptions, according to the following description:

```
previousPSR ← PSR
if ( PSR<Mode> EQ User ) then
    PSR<Mode> ← kernel
    tp ← (IKSP - TrapFrameLength)! Establish trap pointer
else
    tp ← (sp - TrapFrameLength)! Establish trap pointer
endif
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntRa(tp) ← ra
TrIntGp(tp) ← gp
TrIntA0(tp) ← a0
TrIntA1(tp) ← a1
TrIntA2(tp) ← a2
TrIntA3(tp) ← a3
TrFir(tp) ← ExceptionPC
TrPsr(tp) ← previousPSR
sp ← tp
RESTART_ADDRESS ← MEM_MGMT_ENTRY
fp ← sp
```



```
gp ← KGP
a0 ← 1 if store; 0 if load
a1 ← faulting virtual address
a2 ← previousPSR<Mode>
a3 ← previousPSR
```

All other general-purpose registers must be preserved across the memory management exception dispatch.

If the kernel can resolve the fault, it uses the `rfe` instruction to restart the faulting thread, thus reissuing the instruction that faulted. Otherwise, the kernel raises the appropriate exception.

F.4.1.6 System Service Calls

System service calls are initiated from both user and kernel modes via the `callsys` instruction. The privileged `retsys` instruction returns from a system service back to the caller. The `callsys` and `retsys` instructions are described in Sections F.5.2.3 and F.5.1.21, respectively.

F.4.1.7 General Exceptions

General exceptions are those exceptions, other than memory management exceptions and system service call exceptions, that can be raised by hardware or software. All general exceptions are handled in approximately the same manner in the PALcode and in exactly the same manner in the lowest level kernel exception dispatch.

The following exceptions are grouped together as general exceptions:

- Arithmetic exceptions
- Unaligned access exceptions
- Illegal instruction exceptions
- Invalid (non-canonical virtual) address exceptions
- Software exceptions
- Breakpoints
- Subsetted IEEE instruction exceptions

A general exception builds a trap frame on the kernel stack and populates the exception record within the trap frame and then dispatches to the kernel general exception entry point. The common dispatch for general exceptions is shown in Section F.4.1.7.8.

The differences between each type of exception are the population of the exception record and the meaning of the faulting instruction field within the trap frame. The values for each specific exception are detailed in the sections that follow.

F.4.1.7.1 Arithmetic Exceptions

An arithmetic trap occurs at the completion of the operation that caused the exception. Since several instructions may be in various stages of execution at any point in time, it is possible for multiple arithmetic traps to occur simultaneously. The intervening instructions (after the trigger instruction) are collectively called the *trap shadow*. See Section 4.7.7.3 for information.

The ExceptionPC is written to the TrFir offset of the trap frame. The ExceptionPC written into the trap frame is the virtual address of the first instruction after the trapping instruction that has not yet executed.

Note:

Instructions with illegal operands cause illegal instruction exceptions to be raised only if the processor raises an exception for these operations.

Illegal instruction exceptions are precise; the faulting address written into the trap frame is the virtual address of the instruction that caused the exception.

The PALcode writes the following information into the exception record of the trap frame for an illegal instruction exception, where *er* is the exception record pointer.

```
ErExceptionCode(er) ← STATUS_ILLEGAL_INSTRUCTION
ErNumberParameters(er) ← 0
ErExceptionFlags(er) ← 0
```

F.4.1.7.4 Invalid (Non-Canonical Virtual) Address Exceptions

The PALcode raises a general exception if the PALcode detects an invalid faulting virtual address, that is, a faulting virtual address that is not a canonical longword. The implementation must test for the non-canonical format for both data stream and instruction stream translation buffer fills.

For data stream faults, the faulting address written to the trap frame is the virtual address of the instruction that caused the reference to the invalid address.

Instruction stream invalid addresses present a more difficult problem because the exception address itself is invalid and cannot be properly interpreted by a 32-bit operating system. In the case of instruction stream virtual addresses, the *ra* (return address) register minus 4 (*ra*-4) is written to the faulting address field of the trap frame. The *ra* register is used because it probably yields a sane address within the correct program that faulted. Also, the (*ra*-4) is the most probable faulting address, as the most likely instruction to have caused the fault is: *jsr ra, (rx)*.

The PALcode writes the following information into the exception record of the trap frame for a non-canonical virtual address fault, where *er* is the exception record pointer.

```
ErExceptionCode(er) ← STATUS_INVALID_ADDRESS
ErExceptionInformation<0>(er) ← 1 if store; 0 otherwise
ErExceptionInformation<1>(er) ← invalid va<63..32>
ErExceptionInformation<2>(er) ← invalid va<31..0>
ErNumberParameters(er) ← 3
ErExceptionFlags(er) ← 0
ErExceptionRecord(er) ← 0
```

F.4.1.7.5 Software Exceptions

Software may raise exceptions by using the unprivileged *gentrap* (generate trap) instruction. The *gentrap* instruction is used to raise exceptions recognized (possibly) in user-mode software for conditions such as divide by zero. (The Alpha architecture does not provide an integer divide instruction; division is accomplished by specialized divide routines.)

The *gentrap* instruction takes a single parameter that is preserved but not interpreted by the PALcode. The *gentrap* parameter is written into the exception record where it is interpreted by the kernel exception handler. *Gentrap* uses the `STATUS_ALPHA_GENTRAP` status as an exception code. The kernel exception dispatcher interprets the *gentrap* parameter to determine the appropriate Windows NT Alpha status to raise to the currently executing thread.

The faulting address for a gentrap exception is the virtual address of the executed gentrap instruction.

The PALcode writes the following information into the exception record for a gentrap instruction, where *er* is the exception record pointer:

```

ErExceptionCode(er) ← STATUS_ALPHA_GENTRAP
ErExceptionInformation<0>(er) ← gentrap parameter
                               (a0<31..0> upon execution of gentrap)
ErExceptionInformation<1>(er) ← gentrap parameter
                               (a0<63..32> upon execution of gentrap)
ErNumberParameters(er) ← 2
ErExceptionFlags(er) ← 0
ErExceptionRecord(er) ← 0

```

F.4.1.7.6 Breakpoints and Debugger Support

There are several breakpoint instructions and each raises a general exception. Several of these breakpoints are implemented to support the kernel debugger and are essentially special subroutine calls. The exact semantics of these calls are not important to the PALcode; all breakpoints are handled in the same manner and are distinguished only by the breakpoint type that is written into the exception record.

All breakpoints are implemented as unprivileged PALcode instructions, which allows the kernel to decide whether the breakpoint can be taken in the current mode.

Table F–15 lists the breakpoint mnemonics and their corresponding breakpoint types:

Table F–15: Breakpoint Types

Mnemonic	Type	Description
bpt	USER_BREAKPOINT	User breakpoint
kbpt	KERNEL_BREAKPOINT	Kernel breakpoint
callkd	Passed in v0	Call kernel debugger

The faulting instruction address for all breakpoints is the virtual address of the breakpoint instruction.

PALcode completes the exception record for breakpoints as follows, where *er* is the exception record pointer:

```

ErExceptionCode(er) ← STATUS_BREAKPOINT
ErExceptionInformation<0>(er) ← breakpoint type
ErNumberParameters(er) ← 1
ErExceptionFlags(er) ← 0
ErExceptionRecord(er) ← 0

```

F.4.1.7.7 Subsetted IEEE Instruction Exceptions

Floating-point instructions are always enabled. Therefore, FEN (floating enable) faults are not supported.

Hardware Implementation Note:

Windows NT Alpha requires implementation of IEEE floating-point in each processor implementation. The PALcode raises an illegal instruction exception for any subsetted IEEE floating-point instruction — that is, for any IEEE floating-point instruction not implemented in hardware.

VAX floating-point format is not supported.

F.4.1.7.8 General Exceptions: Common Operations

The common operations for all general exceptions are as follows.

```
previousPSR ← PSR
if ( PSR<Mode> EQ User ) then
    PSR<Mode> ← kernel
    tp ← (IKSP - TrapFrameLength)! Establish trap pointer
else
    tp ← (sp - TrapFrameLength) ! Establish trap pointer
endif
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntGp(tp) ← gp
TrIntRa(tp) ← ra
TrIntA0(tp) ← a0
TrIntA1(tp) ← a1
TrIntA2(tp) ← a2
TrIntA3(tp) ← a3
TrPsr(tp) ← previousPSR
TrFir(tp) ← ExceptionPC
sp ← tp
RESTART_ADDRESS ← GENERAL_ENTRY
fp ← sp
gp ← KGP
a0 ← tp + TrExceptionRecord! pointer to exception record
a3 ← previousPSR
```

All other general-purpose registers must be preserved across the general exception dispatch.

F.4.1.8 Panic Exceptions

Severe problems produce panic exceptions. Severe problems are not recoverable; the operating system cannot continue executing normally. Panic exception handling shuts down the machine in a controlled manner that assists in debugging the problem. With the exception of hardware errors, panic exceptions are not expected to occur in the production operating system.

The PALcode raises a panic exception to the kernel and describes the condition that causes the panic with a bugcheck code. When the kernel receives a panic exception, it enters the kernel debugger if it is enabled.

The classes of panic exceptions are:

- Kernel stack corruption
- Unexpected exceptions in PALcode

F.4.1.8.1 Kernel Stack Corruption

The PALcode can recognize the following types of kernel stack corruption: invalid kernel stack, kernel stack overflow, and kernel stack underflow. The kernel stack for an executing thread must always be valid. The PALcode raises a panic exception if the processor faults when accessing the kernel stack and the page tables indicate that the kernel stack address is not valid. The PALcode may also check for kernel stack underflow and overflow and raise a panic exception if either condition is detected.

The kernel stack is the two pages of virtual address space below the IKSP for a thread, where the IKSP points to the byte beyond the top of the two pages. When raising a kernel stack corruption exception, the PALcode sets the bugcheck code to PANIC_STACK_SWITCH.

F.4.1.8.2 Unexpected Exceptions

The PALcode may raise a panic exception when it detects an unexpected condition caused by PALcode. Such unexpected conditions are implementation dependent. It is anticipated that those conditions indicate a bug in the PALcode or that the processor is no longer executing correctly. The PALcode raises the bugcheck code TRAP_CAUSE_UNKNOWN.

F.4.1.8.3 Panic Exception Trap Frame and Dispatch

The PALcode builds a trap frame for the kernel before it dispatches. The PALcode also fills in the exception record that exists within the trap frame.

The PALcode attempts to maintain all possible register state in order to assist in debugging.

The PALcode performs the following operations when dispatching a panic exception to the kernel:

```
previousPSR ← PSR
if ( PSR<Mode> EQ User ) then
    PSR<Mode> ← Kernel
endif
panicStack ← PcPanicStack(PCR)      ! Get the panic stack
tp ← (panicStack - TrapFrameLength)! Allocate trap frame
                                   !   on panic stack

TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntGp(tp) ← gp
TrIntRa(tp) ← ra
TrIntA0(tp) ← a0
TrIntA1(tp) ← a1
TrIntA2(tp) ← a2
TrIntA3(tp) ← a3
TrPsr(tp) ← previousPSR
TrFir(tp) ← ExceptionPC
sp ← tp
fp ← sp
gp ← KGP
a0 ← NT bugcheck code
```

```

a1 ← Exception address
a2, a3, a4 ← Bugcheck parameters
RestartAddress ← PANIC_ENTRY

```

All other general-purpose registers must be preserved across the panic exception dispatch.

F.4.2 Interrupts

The PALcode supports two software interrupt levels and an implementation-specific limit of hardware interrupt sources. The Windows NT Alpha PALcode supports eight levels of interrupt priority known as interrupt request levels (IRQL). The supported IRQLs are numbered 0–7.

The platform independence of interrupt dispatch is accomplished via three tables: Interrupt Level Table, Interrupt Mask Table, and Interrupt Dispatch Table.

F.4.2.1 Interrupt Level Table (ILT)

The Interrupt Level Table consists of eight entries, indexed 0–7. The index values and symbols for the entries are described in Table F–3. Each table entry corresponds to an IRQL by its index within the table. The value of each entry is an enable value that indicates which interrupt sources are to be enabled within the processor for the corresponding IRQL. One full longword is reserved for each table entry. The interpretation of the bits within the enable mask is processor specific.

Implementation Note (Software):

The Interrupt Level Table is probably the most important optional set of data that can be cached within the processor. Implementations should consider implementing a PALcode instruction that causes the ILT to be reread and recached within the processor. Some processors may have an effectively hardwired ILT. In such a case, the HAL has no influence over which interrupts are enabled for each IRQL.

F.4.2.2 Interrupt Mask Table (IMT)

The Interrupt Mask Table relates a mask value of requested interrupts to both an interrupt vector and a synchronization IRQL. The table resolves implicit interrupt priorities because only one interrupt vector can be assigned for each request mask. The IMT is divided into two sub-tables as described in Table F–16.

Table F–16: Interrupt Mask Table (IMT)

Index Range	Interrupt Source Description
0–3	Software (2 sources)
4–131	Hardware

Each entry in the table is a longword that consists of two word values: the interrupt vector number and the synchronization level. The use of the software portion of the table is strictly defined and consistent across all processor implementations.

Implementation Note:

In an implementation, the relation between pending interrupts and their interrupt vectors and synchronization levels may be hardwired. In that case, the IMT is not used and the HAL is not able to influence the setting of priority or assignment of interrupts.

The software entries are used only if no hardware interrupts are pending. The entries must be initialized so that deferred procedure call (DPC) software interrupts are higher priority than asynchronous procedure call (APC) software interrupts. The expected initialization of the software portion of the IMT is defined in Table F-17.

Table F-17: Software Entries of the IMT

Index	Synchronization Level	Vector
0	PASSIVE_LEVEL = 0	Passive release vector
1	APC_LEVEL = 1	APC dispatch vector
2	DISPATCH_LEVEL = 2	DPC dispatch vector
3	DISPATCH_LEVEL = 2	DPC dispatch vector

The hardware portion of the IMT is designed for flexible use. Each implementation must define a relation f that defines a mapping of requested and enabled hardware interrupt sources to entries in the IMT. The relation f is implementation specific, but f must be a function in the mathematical sense (for each input there is a single unambiguous result). All interrupts other than software interrupts are considered hardware interrupts. Hardware interrupts can include external interrupt signals, performance counter interrupts, and correctable read interrupts.

F.4.2.3 Interrupt Dispatch Table (IDT)

The Interrupt Dispatch Table (IDT) has an entry for each possible interrupt vector. The possible interrupt vectors are in the range 0-255. Each entry is a longword pointer, which is the virtual address of the interrupt dispatch routine for the vector that corresponds to the index of the entry within the table. The PALcode does not read or write the IDT; it is maintained and used entirely by the kernel and HAL.

F.4.2.4 Interrupt Dispatch

Interrupt dispatch within the PALcode goes through the following steps:

```
! Mask of requested (irr) and enabled (ier) interrupt sources:
irm ← irr AND ier
! Retrieve value from interrupt mask table:
CASE
  Hardware Interrupt Pending :
    index = f(irm)
    sirql ← (IMT<{index*4}>)<Synchronization IRQL>
    vector ← (IMT<{index*4}>)<InterruptVector>
  Software Interrupt Pending:
    sirql ← (IMT<{irm*4}>)<Synchronization IRQL>
    vector ← (IMT<{irm*4}>)<InterruptVector>
```

```

        Otherwise:
            Passive release, restart execution
    ENDCASE
    Set processor to sirql IRQL
    if ( processor interrupt ) then
        { acknowledge the interrupt }
    endif

```

Once synchronization level has been set and the interrupt service routine has been determined, the PALcode builds a trap frame and dispatches to the kernel interrupt exception handler passing in the interrupt vector.

In the case of software interrupts:

```

previousPsr ← PSR
if ( PSR<Mode> EQ User ) then
    PSR<Mode> ← Kernel
    tp ← (IKSP - TrapFrameLength) ! Establish trap pointer
else
    tp ← (sp - TrapFrameLength)   ! Establish trap pointer
endif
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntGp(tp) ← gp
TrIntA0(tp) ← a0
TrIntA1(tp) ← a1
TrIntA2(tp) ← a2
TrIntA3(tp) ← a3
TrFir(tp) ← ExceptionPC
TrPsr(tp) ← previousPSR
TrIntRa(tp) ← ra
sp ← tp
fp ← sp
gp ← KGP
a0 ← interrupt vector
a1 ← PCR
a2 ← synchronization IRQL
a3 ← previousPSR
RestartAddress ← INTERRUPT_ENTRY

```

In the case of hardware interrupts:

```

PreviousPSR ← PSR
if ( PSR<Mode> EQ User ) then
    PSR<Mode> ← Kernel
    tp ← (IKSP - TrapFrameLength) ! Establish trap pointer
else
    tp ← (sp - TrapFrameLength)   ! Establish trap pointer
endif
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntGp(tp) ← gp
TrIntA0(tp) ← a0
TrIntA1(tp) ← a1
TrIntA2(tp) ← a2

```

```

TrIntA3(tp) ← a3
TrFir(tp) ← ExceptionPC
TrPsr(tp) ← previousPSR
TrIntRa(tp) ← ra
sp ← tp
fp ← sp
gp ← KGP
a0 ← interrupt vector
a1 ← PCR
a2 ← synchronization IRQL
a3 ← previousPSR
RestartAddress ← INTERRUPT_ENTRY

```

All other general-purpose register values must be preserved across interrupt dispatch.

The kernel uses the rfe instruction to restart the interrupted code sequence.

F.4.2.5 Interrupt Acknowledge

Interrupts are acknowledged according to their origin. Internal processor interrupts, such as software interrupts and performance counters, are acknowledged by the PALcode. System-level interrupts are acknowledged in the native interrupt dispatch routines.

F.4.2.6 Synchronization Functions

The swpirql, di, and ei instructions allow the kernel to affect the processor's current interrupt enable state:

- Swpirql swaps the current interrupt request level (IRQL) of the processor. Swpirql takes the new IRQL as a parameter and returns the previous IRQL.
- Di disables all interrupts without changing the current IRQL.
- Ei enables interrupts at the currently set IRQL.

Those instructions and the existence of the interrupt enable bit in the PSR are used as a global interrupt enable for all interrupts.

F.4.2.7 Software Interrupt Requests

The PALcode includes the software interrupt request register (SIRR), an architected internal processor register, for controlling software interrupt requests. The PALcode also includes two instructions, ssir and csir, to control the state of the SIRR register.

The format of the SIRR is shown in Figure F-7 and the fields are defined in Table F-18.

Figure F–7: Software Interrupt Request Register

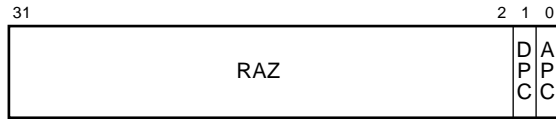


Table F–18: Software Interrupt Request Register Fields

Field	Type	Description
DPC	RW	DPC software interrupt requested
APC	RW	APC software interrupt requested

The `ssir` and `csir` instructions affect the state of software interrupt requests.

The `ssir` instruction sets software interrupt requests by taking as a parameter the interrupt request levels to be set. Setting the appropriate bit in `SIRR` indicates that the corresponding software interrupt is requested. The `csir` instruction clears software interrupt requests by taking as a parameter the interrupt request level to be cleared. Clearing the appropriate bit in `SIRR` indicates that the corresponding software interrupt request has been cleared.

F.4.3 Machine Checks

Machine checks are initiated when the hardware detects a hardware error condition. However, machine checks are not the only way that detected hardware errors are reported. Hardware error conditions can be reported from three sources:

- At the pin level. Hardware may choose to signal errors via hardware interrupts. PALcode delivers such hardware error interrupts to the kernel as standard interrupts, where they may be hooked by the HAL for system-specific processing. Such interrupts are not processed by the PALcode as machine checks and are not described in this section.
- From an implementation-dependent internal error interrupt. It is an implementation decision whether to deliver such an interrupt as a standard interrupt or as a machine check. The processing of an interrupt that is delivered as a machine check is described in this section.
- At the machine check hardware vector. Hardware errors that are signaled by the processor through a specific machine check hardware vector are considered machine checks and are described in this section.

The machine check condition may be correctable or uncorrectable. If uncorrectable, the hardware may choose to retry the operation that returned the error.

The PALcode recognizes the following types of machine checks:

- Correctable errors
- Uncorrectable errors
- Catastrophic errors

F.4.3.1 Correctable Errors

Processor correctable errors are data errors that are detected by the processor and can be reliably corrected. System correctable errors are detected and corrected by the system hardware; incorrect data is not read into the processor.

Correctable errors are maskable by the MCES internal processor register (Figure F-8). It is recommended that correctable errors be disabled during PALcode initialization and subsequently be explicitly enabled by the HAL. Correctable errors are delivered from the PALcode to allow the HAL to log the errors. The PALcode builds a logout frame with per-processor information that assists the HAL in logging the error.

F.4.3.2 Uncorrectable Errors

Uncorrectable errors from the processor are detected by the processor and exhibit data errors that cannot be reliably corrected. Actual processor uncorrectable errors are defined by the processor implementation. Uncorrectable errors from the system are detected but not corrected by the system hardware.

Although uncorrectable errors are likely also to be unrecoverable, a mechanism exists in the exception record to allow one or more retries when appropriate. The HAL controls the retry count. For example, a parity error in the I-cache, although uncorrectable, may disappear after an operation retry.

The machine check exception is raised to the HAL to allow per-platform error handling. Uncorrectable errors are delivered immediately upon detection. The PALcode creates a logout frame with per-processor information to assist the HAL in handling the error condition.

F.4.3.3 Machine Check Error Handling

The general model for machine check handling has the following flow:

1. The PALcode corrects the error, if possible.
2. The PALcode sets the machine to a known state from which restart is possible.
3. The PALcode builds a logout frame describing the detected error.
4. The PALcode sets processor IRQL appropriately (see below).
5. The PALcode dispatches a general exception to the kernel.
6. In the case of a catastrophic error, PALcode returns control to the firmware, as described in Section F.4.3.4.

The machine check error summary (MCES) register, Figure F-8, indicates and controls the current state of the machine check handler for the processor. Table F-19 describes the MCES register.

The `draina` instruction, when coupled with appropriate implementation-specific native code, can allow software to force completion of all previously executed instructions, such that the previous instructions cannot cause machine checks to be signaled while any instructions subsequent to the `draina` are executed.

F.4.3.4 Catastrophic Errors

Although particular catastrophic conditions are specific to the processor implementation, such conditions indicate that the machine is left in a state where execution cannot be reliably restarted. They also indicate that the hardware cannot be trusted to execute properly or the state of data within the system cannot be determined.

An example of a catastrophic condition is a machine check taken while machine check handling is in progress, as indicated by a set `MCK` bit in the `MCES` register. Taking a machine check while in the PALcode environment is also considered catastrophic. In those cases, control is returned to the firmware as follows:

1. Further machine check acknowledgement is turned off and a logout frame is generated.
2. The restart block is verified:
 - If the restart block is good, the current state in the restart block is saved, the previous state is restored, and control is returned to the firmware at the restart address.
 - If the restart block is bad, the alternate path is used to re-execute the previous PALcode image at its entry address. See Section F.6.2.1

F.5 PALcode Instruction Descriptions

The PALcode instructions generally follow the Windows NT Alpha calling standard. Arguments are passed in the argument (`a0–a5`) registers and return values are returned in the value (`v0`) register. The PALcode instructions also incorporate the following conventions into their own calling standard:

- Unless specific temporary registers are required, only the argument registers `a0–a5` are considered volatile.
- Generally, all parameters are passed in registers.

The argument registers are used as volatile registers because often they contain parameters to the PALcode instructions. In strict adherence to the calling standard, the temporary registers `t0–t12` could also be considered volatile in the PALcode instructions, but they are not. The temporary registers are not considered necessarily volatile because PALcode instructions generally do not need more free registers. Further, it is convenient in assembly language, from which the PALcode instructions are most frequently called, to be able to assume that temporary registers are preserved across the PALcode instruction.

All parameters to the PALcode instructions are passed in registers. If the number of parameters exceeds the available number of argument registers, additional temporary registers are used as arguments. This precludes the need for callers to build an appropriate stack frame for PALcode instructions with more than six parameters.

The `RESTART_ADDRESS` register indicates the next execution address when the PALcode exits. Upon entry to each of the PALcode instructions, the `RESTART_ADDRESS` register is considered to contain the address of the instruction immediately following the PALcode instructions.

A range of privileged PALcode instructions is reserved for processor-implementation-specific PALcode instructions that allow specialized communication between the HAL and the PALcode.

Note: The *Operation* part of the PALcode instruction descriptions is shown as an ordered sequence of instructions. The instructions in the sequence may be reordered as long as the results of the sequence of instructions are not altered. In particular, if an instruction *j* is listed subsequent to an instruction *i* and *i* writes any data that is used by *j*, then *i* must be executed before *j*.

F.5.1 Privileged PALcode Instructions

Table F–21 summarizes the privileged PALcode instructions.

Table F–21: Privileged PALcode Instruction Summary

Mnemonic	Description
<code>csir</code>	Clear software interrupt request
<code>dalnfix</code>	Disable alignment fixups
<code>di</code>	Disable interrupts
<code>draina</code>	Drain aborts
<code>dtbis</code>	Data translation buffer invalidate single
<code>ealnfix</code>	Enable alignment fixups
<code>ei</code>	Enable interrupts
<code>halt</code>	Halt the processor
<code>initpal</code>	Initialize the PALcode
<code>initpcr</code>	Initialize PCR data
<code>rdcounters</code>	Read PALcode event counters
<code>rdirq</code>	Read current IRQ
<code>rdksp</code>	Read initial kernel stack
<code>rdmces</code>	Read machine check error summary
<code>rdpcr</code>	Read processor control region address
<code>rdpsr</code>	Read processor status register
<code>rdstate</code>	Read internal processor state
<code>rdthread</code>	Read the current thread value
<code>reboot</code>	Transfer to console or previous PALcode environment

Table F–21: Privileged PALcode Instruction Summary (Continued)

Mnemonic	Description
restart	Restart the processor
retsys	Return from system service call
rfe	Return from exception
ssir	Set software interrupt request
swpctx	Swap privileged thread context
swpirql	Swap IRQL
swpksp	Swap initial kernel stack
swppal	Swap PALcode
swpprocess	Swap privileged process context
tbia	Translation buffer invalidate all
tbim	Translation buffer invalidate multiple
tbimasn	Translation buffer invalidate multiple for ASN
tbis	Translation buffer invalidate single
tbisasn	Translation buffer invalidate for single ASN
wrentry	Write system entry
wrmces	Write machine check error summary
wrperfmon	Write performance monitoring values

F.5.1.1 Clear Software Interrupt Request

Format:

csir

! PALcode format

Operation:

```
{ a0 = Software interrupt requests to clear }  
  
if ( PSR<Mode> EQ User ) then  
    { Initiate illegal instruction exception }  
endif  
if ( a0<1> EQ 1 ) then  
    SIRR<DPC> ← 0  
endif  
if ( a0<0> EQ 1 ) then  
    SIRR<APC> ← 0  
endif
```

GPR State Change:

a0–a5 are UNPREDICTABLE

IPR State Change:

SIRR ← 0 according to a0

Exceptions:

Illegal Instruction
Machine Checks

Description:

The csir instruction clears the specified bit in the SIRR internal processor register, depending on the contents of a0. See Section F.4.2.7.

F.5.1.2 Disable Alignment Fixups

Format:

dalnfix

! PALcode format

Operation:

```
if ( PSR<Mode> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ Implementation-specific state is set to generate alignment fault }
{ exceptions and to prevent alignment fault fixups by the PALcode }
```

GPR State Change:

None

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The dalnfix instruction disables alignment fault fixups in PALcode and generates alignment fault exceptions whenever an alignment fault occurs. After dalnfix is executed on a processor, all alignment faults on that processor are not fixed-up by PALcode and alignment fault exceptions are dispatched to the kernel until the ealnfix instruction is executed on that processor.

F.5.1.3 Disable All Interrupts

Format:

di

! PALcode format

Operation:

```
if ( PSR<Mode> EQ User ) then
    { Initiate illegal instruction exception }
endif
PSR<IE> ← 0
```

GPR State Change:

None

IPR State Change:

PSR<IE> ← 0

Exceptions:

Illegal Instruction
Machine Checks

Description:

The di instruction disables all interrupts by clearing the interrupt enable bit (IE) in the PSR internal processor register. The IRQL field is unaffected. Interrupts may be re-enabled via the ei instruction.

F.5.1.4 Drain All Aborts Including Machine Checks

Format:

draina

! PALcode format

Operation:

```
if ( PSR<Mode> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ Implementation-specific drain }
```

GPR State Change:

None

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The draina instruction facilitates the draining of all aborts, including machine checks, from the current processor. When coupled with the appropriate implementation-specific native code, draina can help guarantee that no abort is signaled for an instruction issued before the draina while any instruction issued subsequent to the draina is executing.

F.5.1.5 Data Translation Buffer Invalidate Single

Format:

dtbis ! PALcode format

Operation:

```
{ a0 = Virtual address of translation to invalidate }  
  
if ( PSR<MODE> EQ User ) then  
    { Initiate illegal instruction exception }  
endif  
{ Invalidate all translations in the data stream for the }  
{ virtual address in a0 }
```

GPR State Change:

a0–a5 are UNPREDICTABLE.

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The dtbis instruction invalidates a single data stream translation. The translation for the virtual address in a0 must be invalidated in all data translation buffers and in all virtual data caches.

F.5.1.6 Enable Alignment Fixups

Format:

ealnfix

! PALcode format

Operation:

```
if ( PSR<Mode> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ Implementation-specific state is set to fix up alignment fault }
{   by the PALcode }
```

GPR State Change:

None

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The ealnfix instruction enables alignment fault fixups in PALcode and prevents alignment fault exceptions. After ealnfix is executed on a processor, all alignment faults on that processor are fixed-up by PALcode and no alignment fault exceptions are dispatched to the kernel until the dalnfix instruction is executed on that processor.

The default state is disabled alignment fixups in PALcode.

F.5.1.7 Enable Interrupts

Format:

ei

! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
PSR<IE> ← 1
```

GPR State Change:

None

IPR State Change:

PSR<IE> ← 1

Exceptions:

Illegal Instruction
Machine Checks

Description:

The ei instruction sets the interrupt enable (IE) bit in the PSR internal processor register, thus enabling those interrupts that are at the appropriate level for the current IRQL field in the PSR.

F.5.1.8 Halt the Operating System by Trapping to Illegal Instruction

Format:

halt

! PALcode format

Operation:

{ Initiate illegal instruction exception }

GPR State Change:

See Section F.4.1.7.3 for illegal instruction exception handling.

IPR State Change:

See Section F.4.1.7.3 for illegal instruction exception handling.

Exceptions:

Illegal Instruction

Description:

The halt instruction forces an illegal instruction exception. See the reboot instruction, Section F.5.1.19, for transferring control to the console or previous PALcode environment.

F.5.1.9 Initialize PALcode Data Structures with Operating System Values

Format:

```
initpal                                     ! PALcode format
```

Operation:

```
{ a0 = Page directory entry (PDE) page, superpage 32 address }
{ a1 = Initial thread value }
{ a2 = Initial TEB value }
{ gp = Kernel global pointer }
{ sp = Initial kernel stack pointer }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
PDR      ← (a0 BIC 8000000016 )
THREAD   ← a1
TEB      ← a2
IKSP     ← sp
KGP      ← gp
PcPalBaseAddress(PCR)      ← PAL_BASE
PcPalMajorVersion(PCR)    ← PalMajorVersion
PcPalMinorVersion(PCR)    ← PalMinorVersion
PcPalSequenceVersion(PCR) ← PalSequenceVersion
PcPalMajorSpecification(PCR) ← PalMajorSpecification
PcPalMinorSpecification(PCR) ← PalMinorSpecification
v0 ← PAL_BASE
```

GPR State Change:

```
v0 ← PAL_BASE
a0–a5 are UNPREDICTABLE.
```

IPR State Change:

```
PDR ← a0
THREAD ← a1
TEB ← a2
IKSP ← sp
KGP ← gp
```

Exceptions:

```
Illegal Instruction
Machine Checks
```

Description:

The `initpal` instruction is called early in the kernel initialization sequence to establish IPR values for the initial thread PDR, THREAD, TEB, and IKSP. The IPR value KGP persists for the life of the system. In addition, `initpal` writes the PALcode version information into the PCR.

On return from the `initpal` instruction, the return value register, `v0`, contains the `PAL_BASE` register, the base address in 32-bit superpage (`kseg0`) format.

F.5.1.10 Initialize Processor Control Region Data

Format:

initpcr ! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif

{ Cache portions of Interrupt Level Table and Processor Control Region }
{ data in implementation-dependent manner }
```

GPR State Change:

a0–a4 are UNPREDICTABLE

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The `initpcr` instruction caches process-specific information, including parts of the Interrupt Level Table (ILT), for use by the PALcode. See Section F.6.1.4 for information on the ILT.

F.5.1.11 Read the Software Event Counters

Format:

rdcounters

! PALcode format

Operation:

```
{ a0 = Pointer to 32-bit superpage address of counter record buffer. }
{   Address must be quadword aligned }
{ a1 = Length of buffer in bytes }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ Dump event counter values to the counter record }
v0 ← status
```

GPR State Change:

v0 ← status
a0–a5 are UNPREDICTABLE.

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

For debug PALcode (see Section F.5.3), rdcnters causes that PALcode to write the state of its internal software event counters into an implementation-specific counter record pointed to by the address passed in the a0 register. For production PALcode, rdcnters returns a status value of zero, indicating that it is not implemented in the current PALcode image.

On return from rdcnters, v0 contains the status as follows:

If v0 = 0 Interface is not implemented.
If v0 ≤ a1 v0 is length of data returned.
If v0 > a1 No data is returned and v0 is length of processor implementation
 counter record.

F.5.1.12 Read the Current IRQL from the PSR

Format:

rdirql

! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
v0 ← PSR<IRQL>
```

GPR State Change:

v0 ← <IRQL>

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The rdirql instruction returns in v0 the contents of the interrupt request level (IRQL) field of the PSR internal processor register.

F.5.1.13 Read Initial Kernel Stack Pointer for the Current Thread

Format:

rdksp

! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
v0 ← IKSP
```

GPR State Change:

v0 ← <IKSP>

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The rdksp instruction returns in v0 the contents of the IKSP (initial kernel stack pointer) internal processor register for the currently executing thread.

F.5.1.14 Read the Machine Check Error Summary Register

Format:

rdmces

! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
v0 ← MCES
```

GPR State Change:

v0 ← MCES

IPR State Change:

none

Exceptions:

Illegal Instruction
Machine Checks

Description:

The rdmces instruction returns in v0 the contents of the machine check error summary (MCES) internal processor register.

F.5.1.15 Read the Processor Control Region Base Address

Format:

rdpcr

! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
v0 ← PCR
```

GPR State Change:

v0 ← PCR

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The rdpcr instruction returns in v0 the contents of the PCR internal processor register (the base address value of the processor control region).

F.5.1.16 Read the Current Processor Status Register (PSR)

Format:

rdpsr

! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
v0 ← PSR
```

GPR State Change:

v0 ← PSR

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The rdpsr instruction returns in v0 the contents of the current PSR (Processor Status Register) internal processor register.

F.5.1.17 Read the Current Internal Processor State

Format:

rdstate

! PALcode format

Operation:

```
{ a0 = Pointer to 32-bit superpage address of state record buffer. }
{   Address must be quadword aligned }
{ a1 = Length of buffer in bytes }
```

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ Dump internal processor state record to processor state buffer }
v0 ← status
```

GPR State Change:

v0 ← status
a0–a5 are UNPREDICTABLE.

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The rdstate instruction writes the internal processor state to the internal processor state buffer pointed to by the address passed in the a0 register. The form and content of the internal processor state buffer are implementation specific.

On return from the rdstate instruction, the return value register, v0, contains the status as follows:

If v0 = 0 Interface is not implemented.
If v0 ≤ a1 v0 is length of data returned.
If v0 > a1 No data is returned and v0 is length of processor implementation counter record.

F.5.1.18 Read the Thread Value for the Current Thread

Format:

rdthread

! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
v0 ← THREAD
```

GPR State Change:

v0 ← THREAD

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The rdthread instruction returns in v0 the contents of the THREAD internal processor register (for the currently executing thread).

F.5.1.19 Reboot — Transfer to Console Firmware

Format:

reboot ! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
RestartBlockPointer ← PcRestartBlock(PCR )
{ If cannot verify restart block, restart previous PALcode }
{ Save general register state in saved state area }
{ Save internal processor register state in saved state area, }
{ includes PAL_BASE }
{ Save implementation-specific data in saved state area }
{ Set the saved state length in restart block }
{ Compute and store Checksum for restart block }
{ Restore previous privileged state }
PAL_BASE ← previous_PAL_BASE.
RESTART_ADDRESS ← PcFirmwareRestartAddress(PCR)
```

GPR State Change:

All registers are UNPREDICTABLE.

IPR State Change:

```
PAL_BASE ← previous_PAL_BASE
RESTART_ADDRESS ← PcFirmwareRestartAddress(PCR)
All other registers are UNPREDICTABLE.
```

Exceptions:

Illegal Instruction
Machine Checks

Description:

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. The PALcode must accomplish two tasks to restore the boot environment: re-establish the boot environment PALcode and restart execution in the boot environment at the Firmware Restart Address.

F.5.1.20 Restart the Operating System from the Restart Block

Format:

restart

! PALcode format

Operation:

```
{ a0 = Pointer to ARC restart block with Alpha saved state area }  
  
if ( PSR<MODE> EQ User ) then  
    { Initiate illegal instruction exception }  
endif  
{ Verify restart block }  
{ if invalid then return to caller }  
RestartBlockPointer ← PcRestartBlock(PCR)  
{ Restore general register state from saved state area }  
{ Restore internal processor register state from saved state area, }  
{ Restore implementation-specific data from saved state area }  
RESTART_ADDRESS ← RbRestartAddress(RestartBlockPointer)
```

GPR State Change:

All registers are UNPREDICTABLE.

IPR State Change:

RESTART_ADDRESS ← RbRestartAddress(RestartBlockPointer)

All other registers are UNPREDICTABLE.

Exceptions:

Illegal Instruction

Machine Checks

Description:

The restart instruction restores saved processor state and resumes execution of the operating system.

F.5.1.21 Return from System Service Call Exception

Format:

retsys

! PALcode format

Operation:

```
{ a0 = Previous PSR }
{ a1 = New software interrupt requests }
{ fp = Pointer to trap frame }
{ v0 = Return status from system service }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
if ( a1<1> EQ 1 ) then
    SIRR<DPC> ← 1
endif
if ( a1<0> EQ 1 ) then
    SIRR<APC> ← 1
endif
TrapFrame ← fp
ra ← TrIntRa(TrapFrame)
gp ← TrIntGp(TrapFrame)
fp ← TrIntFp(TrapFrame)
sp ← TrIntSp(TrapFrame)
RESTART_ADDRESS ← TrFir(TrapFrame)
PSR ← a0
{ Clear lock_flag register }
{ Clear intr_flag register }
```

GPR State Change:

```
ra ← TrIntRa(TrapFrame)
gp ← TrIntGp(TrapFrame)
fp ← TrIntFp(TrapFrame)
sp ← TrIntSp(TrapFrame)
at, t0-t12, a -a5 are UNPREDICTABLE
```

IPR State Change:

```
PSR ← a0
RESTART_ADDRESS ← TrFir(TrapFrame)
SIRR ← a1<1...0>
```

Exceptions:

- Illegal Instruction
- Machine Checks
- Invalid Kernel Stack

Description:

The `retsys` instruction returns from a system service call exception by unwinding the trap frame, clearing the `lock_flag` and `intr_flag` (interrupt flag) registers, and returning to the code stream that was executing when the original exception was initiated. `Retsys` must return to the native code stream; it is illegal for `retsys` to return to the PALcode environment and that must be guaranteed not to happen. In addition, `retsys` accepts a parameter to set software interrupt requests that became pending while the exception was handled.

`Retsys` is similar to the `rfe` instruction, with the following exceptions:

1. `Retsys` need not restore the argument registers `a0–a3` from the trap frame.
2. `Retsys` need not preserve volatile register state.
3. `Retsys` returns to the address in the `ra` register at the point of the `callsys` rather than the faulting instruction address (the `ra` was written to the faulting instruction address by `callsys`).

F.5.1.22 Return from Exception or Interrupt

Format:

rfe

! PALcode format

Operation:

```
{ a0 = Previous PSR }
{ a1 = New software interrupt requests }
{ fp = Pointer to trap frame }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
if ( a1<1> EQ 1 ) then
    SIRR<DPC> ← 1
endif
if ( a1<0> EQ 1 ) then
    SIRR<APC> ← 1
endif

PSR ← a0
TrapFrame ← fp
a0 ← TrIntA0(TrapFrame)
a1 ← TrIntA1(TrapFrame)
a2 ← TrIntA2(TrapFrame)
a3 ← TrIntA3(TrapFrame)
ra ← TrIntRa(TrapFrame)
gp ← TrIntGp(TrapFrame)
fp ← TrIntFp(TrapFrame)
sp ← TrIntSp(TrapFrame)
RESTART_ADDRESS ← TrFir(TrapFrame)

{ Clear lock_flag register }
```

GPR State Change:

```
a0 ← TrIntA0(TrapFrame)
a1 ← TrIntA1(TrapFrame)
a2 ← TrIntA2(TrapFrame)
a3 ← TrIntA3(TrapFrame)
ra ← TrIntRa(TrapFrame)
gp ← TrIntGp(TrapFrame)
fp ← TrIntFp(TrapFrame)
sp ← TrIntSp(TrapFrame)
```

IPR State Change:

PSR \leftarrow a0
RESTART_ADDRESS \leftarrow TrFir(TrapFrame)
SIRR \leftarrow a1<1...0>

Exceptions:

Illegal Instruction
Machine Checks
Invalid Kernel Stack

Description:

The rfe instruction returns from exceptions or interrupts by unwinding the trap frame, clearing the lock_flag register, and returning to the code stream that was executing when the original exception or interrupt was initiated. Rfe must return to the native code stream; it is illegal for rfe to return to the PALcode environment and that must be guaranteed not to happen. In addition, rfe accepts a parameter to set software interrupt requests that became pending while the event was handled.

F.5.1.23 Set Software Interrupt Request

Format:

ssir

! PALcode format

Operation:

```
{ a0 = Software interrupt requests to set }  
  
if ( PSR<MODE> EQ User ) then  
    {Initiate illegal instruction exception }  
endif  
if ( a0<1> EQ 1 ) then  
    SIRR<DPC> ← 1  
endif  
if ( a0<0> EQ 1 ) then  
    SIRR<APC> ← 1  
endif
```

GPR State Change:

a -a5 are UNPREDICTABLE.

IPR State Change:

SIRR ← a0<1...0>

Exceptions:

Illegal Instruction
Machine Checks

Description:

The ssir instruction sets software interrupt requests by setting the appropriate bits in the SIRR internal processor register. See Section F.4.2.7.

F.5.1.24 Swap Thread Context

Format:

swpctx

! PALcode format

Operation:

```
{ a0 = New initial kernel stack va }
{ a1 = New thread address }
{ a2 = New thread environment block pointer }
{ a3 = New address space page frame number (PFN) }
{      or a negative number }
{ a4 = ASN }
{ a5 = ASN_wrap_indicator }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
IKSP ← a0
THREAD ← a1
TEB ← a2
ASN_wrap_indicator ← a5
if ( a3 GE 0 ) then      ! swap address space
    temp ← SHIFT_LEFT( a3, PAGE_SHIFT )
    PDR ← temp
    ASN ← a4
    if ( ASN_wrap_indicator NE 0 ) then
        { invalidate all translations and virtual cache blocks }
        {   for which ASM EQ 0 }
    endif
endif

{   Where:   }
{           2**PAGE_SHIFT = implementation page size }
```

GPR State Change:

a0 a5 are UNPREDICTABLE

IPR State Change:

```
IKSP ← a0
THREAD ← a1
TEB ← a2
PDR ← a3 (possibly)
ASN ← a4 (possibly)
```

Exceptions:

- Illegal Instruction
- Machine Checks

Description:

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.

Swpctx may also swap the address space (or process) for the new thread. If the new thread is in the same process (address space) as the previous thread, the kernel passes a negative value for the page frame number (PFN) in the page directory page, indicating that the address space need not be switched. If the PFN is zero or a positive number, it is used to swap the address space, just as if swpprocess had been executed.

F.5.1.25 Swap the Current IRQL (Interrupt Request Level)

Format:

swpirql

! PALcode format

Operation:

```
{ a0 = New IRQL }  
  
if ( PSR<MODE> EQ User ) then  
    { Initiate illegal instruction exception }  
endif  
v0 ← PSR<IRQL>  
PSR<IRQL> ← a0
```

GPR State Change:

v0 ← PSR<IRQL>
a0–a5 are UNPREDICTABLE.

IPR State Change:

PSR<IRQL> ← a0

Exceptions:

Illegal Instruction
Machine Checks

Description:

The swpirql instruction swaps the current IRQL field in the PSR internal processor register for the specified new IRQL, setting the processor so that only interrupts permitted by the new IRQL are enabled. Swpirql updates the IRQL field and returns in v0 the previous IRQL.

F.5.1.26 Swap the Initial Kernel Stack Pointer (IKSP) for the Current Thread

Format:

```
swpksp                                     ! PALcode format
```

Operation:

```
{ a0 = New IKSP }  
  
if ( PSR<MODE> EQ User ) then  
    { Initiate illegal instruction exception }  
endif  
v0 ← IKSP  
IKSP ← a0
```

GPR State Change:

```
v0 ← IKSP  
a0–a5 are UNPREDICTABLE.
```

IPR State Change:

```
IKSP ← a0
```

Exceptions:

```
Illegal Instruction  
Machine Checks
```

Description:

The swpksp instruction returns in v0 the value of the previous IKSP internal processor register and writes a new IKSP for the currently executing thread.

F.5.1.27 Swap the Currently Executing PALcode

Format:

swppal

! PALcode format

Operation:

```
{ a0 = Physical base address of new PALcode }
{ a1-a5 = Arguments to the new PALcode environment }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ load processor-dependent parameters }
{ jump to address in a0 as a physical address in }
{ the PALcode environment }
```

GPR State Change:

a0 and t0–t12 are UNPREDICTABLE or contain processor-dependent parameters.

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The swppal instruction swaps the currently executing PALcode by transferring to the base address of the new PALcode image (provided in a0) in the PALcode environment.

F.5.1.28 Swap Process Context (Swap Address Space)

Format:

swpprocess

! PALcode format

Operation:

```
{ a0 = Page frame number (PFN) of new PDR }
{ a1 = Address space number (ASN) of new process }
{ a2 = Address space number wrap indicator (ASN_wrap_indicator): }
{     zerp = no wrap }
{     nonzero = wrap }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
temp ← SHIFT_LEFT( a0, PAGE_SHIFT )
PDR ← temp
ASN ← a1
if ( ASN_wrap_indicator NE 0 ) then
    { Invalidate all translations and virtual cache blocks }
    {   for which ASM EQ 0 }
endif

{   Where: }
{     2**PAGE_SHIFT = implementation page size }
```

GPR State Change:

a0–a5 are UNPREDICTABLE..

IPR State Change:

PDR ← a0
ASN ← a1

Exceptions:

Illegal Instruction
Machine Checks

Description:

The swpprocess instruction swaps the privileged process context by changing the address space for the currently executing thread. The address space change is accomplished by establishing a new PDR and ASN. If the ASN_wrap_indicator passed in a2 is nonzero, swpprocess causes invalidation of all translation buffer entries and virtual cache blocks that have a clear address space match (ASM) bit.

F.5.1.29 Translation Buffer Invalidate All

Format:

tbia

! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ Invalidate all translations and virtual cache blocks }
{   within the processor }
```

GPR State Change:

a –a5 are UNPREDICTABLE.

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The tbia instruction invalidates all translations and virtual cache blocks within the processor.

F.5.1.30 Translation Buffer Invalidate Multiple

Format:

tbim ! PALcode format

Operation:

```
{ a0 = Pointer to array of virtual addresses to invalidate }
{ a1 = Number of virtual addresses to invalidate }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ Invalidate translations for virtual addresses pointed to in a0 for }
{ the number of entries in a1. Invalidate in all translation }
{ buffers and all virtual caches }
```

GPR State Change:

a0–a5 are UNPREDICTABLE.

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The tbim instruction invalidates multiple virtual translations for the current ASN. The translations for the virtual address must be invalidated in all processor translation buffers and virtual caches.

F.5.1.31 Translation Buffer Invalidate Multiple for ASN

Format:

tbimasn ! PALcode format

Operation:

```
{ a0 = Pointer to array of virtual addresses to invalidate }
{ a1 = Number of virtual addresses to invalidate }
{ a2 = Address space number (ASN) }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ Invalidate translations for the virtual addresses in the array }
{ pointed to in a0, for the number of entries in a1, that match the }
{ ASN in a2. Invalidate in all translation buffers and virtual caches }
```

GPR State Change:

a0–a5 are UNPREDICTABLE.

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The tbimasn instruction invalidates multiple virtual translations for a specified ASN. The translations for the virtual addresses must be invalidated in all processor translation buffers and virtual caches.

F.5.1.32 Translation Buffer Invalidate Single

Format:

tbis

! PALcode format

Operation:

```
{ a0 = Virtual address of translation to invalidate }  
  
if ( PSR<MODE> EQ User ) then  
    { Initiate illegal instruction exception }  
endif  
{ Invalidate all translations for the virtual address in a0, }  
{ invalidate in all translation buffers and all virtual caches }
```

GPR State Change:

a0–a5 are UNPREDICTABLE.

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

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.

F.5.1.33 Translation Buffer Invalidate Single for ASN

Format:

tbisasn

! PALcode format

Operation:

```
{ a0 = Virtual address of translation to invalidate }
{ a1 = Address space number (ASN) }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
{ Invalidate the translation for the virtual address in a0 }
{ that matches the ASN in a1. The translation must be invalidated }
{ in all translation buffers and virtual caches }
```

GPR State Change:

a0–a5 are UNPREDICTABLE.

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The tbisasn instruction invalidates a single virtual translation for a specified address space number. The translation for the passed virtual address must be invalidated in all processor translation buffers and virtual caches.

F.5.1.34 Write Kernel Exception Entry Routine

Format:

wrentry

! PALcode format

Operation:

```
{ a0 = Address of exception entry routine, 32-bit }
{   superpage address   }
{ a1 = Exception class value }
```

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
case a1 begin
    0:
        PANIC_ENTRY ← a0
        break;
    1:
        MEM_MGMT_ENTRY ← a0
        break;
    2:
        INTERRUPT_ENTRY ← a0
        break;
    3:
        SYSCALL_ENTRY ← a0
        break;
    4:
        GENERAL_ENTRY ← a0
        break;
    otherwise:
        { Initiate panic exception }
endcase;
```

GPR State Change:

a0–a5 are UNPREDICTABLE.

IPR State Change:

*_ENTRY ← a0

Exceptions:

Illegal Instruction
Machine Checks
Panic Exception

Description:

The wrentry instruction provides the registry of exception handling routines for the exception classes. The address in a0 is registered for the exception class corresponding to the exception class value in a1. The kernel must use wrentry to register an exception handler for each of the exception classes. The relationship between the exception classes and class values is shown in Table F-22.

Table F-22: Exception Class Values

Exception Class	Value
Panic exceptions	0
Memory management exceptions	1
Interrupt exceptions	2
System service call exceptions	3
General exceptions	4

F.5.1.35 Write the Machine Check Error Summary Register

Format:

wrmces

! PALcode format

Operation:

```
{a0 = New values for the machine check error }
{      summary (MCES) register. }

if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif
v0 ← MCES
MCES<DMK> ← a0<5>
MCES<DSC> ← a0<4>
MCES<DPC> ← a0<3>
if ( a0<2> EQ 1 ) then
    MCES<PCE> ← 0
endif
if ( a0<1> EQ 1 ) then
    MCES<SCE> ← 0
endif
if( a0<0> EQ 1 ) then
    MCES<MCK> ← 0
endif
```

GPR State Change:

v0 ← previous MCES

IPR State Change:

MCES ← a0

Exceptions:

Illegal Instruction
Machine Checks

Description:

The wrmces instruction writes new values for the MCES internal processor register and returns in v0 the previous values of that register.

F.5.1.36 Write Performance Counter Interrupt Control Information

Format:

wrperfmon

! PALcode format

Operation:

```
if ( PSR<MODE> EQ User ) then
    { Initiate illegal instruction exception }
endif

{ a0 - a5 contain implementation-specific input values }
```

GPR State Change:

v0 ← implementation-dependent value
a0–a5 are UNPREDICTABLE

IPR State Change:

None

Exceptions:

Illegal Instruction
Machine Checks

Description:

The wrperfmon instruction controls any performance monitoring mechanisms in the processor and PALcode. The wrperfmon instruction arguments and actions are chip dependent, and when defined for an implementation, are described in Appendix E.

F.5.2 Unprivileged PALcode Instructions

Table F-23: Unprivileged PALcode Instruction Summary

Mnemonic	Description
bpt	Breakpoint trap
callkd	Call kernel debugger
callsys	Call system service
gentrap	Generate trap
imb	Instruction memory barrier
kbpt	Kernel breakpoint trap
rdteb	Read thread environment block pointer

F.5.2.1 Breakpoint Trap (Standard User-Mode Breakpoint)

Format:

bpt

! PALcode format

Operation:

See Sections F.4.1.7.8 and F.4.1.7.6

GPR State Change:

See Sections F.4.1.7.8 and F.4.1.7.6

IPR State Change:

See Sections F.4.1.7.8 and F.4.1.7.6

Exceptions:

Machine Checks

Kernel Stack Invalid

Description:

The bpt instruction raises a breakpoint general exception to the kernel, setting a USER_BREAKPOINT breakpoint type.

F.5.2.2 Call Kernel Debugger

Format:

callkd

! PALcode format

Operation:

{v0 = Type of breakpoint }
See Sections F.4.1.7.8 and F.4.1.7.6

GPR State Change:

See Sections F.4.1.7.8 and F.4.1.7.6

IPR State Change:

See Sections F.4.1.7.8 and F.4.1.7.6

Exceptions:

Machine Checks
Kernel Stack Invalid

Description:

The callkd instruction raises a breakpoint general exception to the kernel, setting the breakpoint type with the value supplied in v0. The callkd instruction implements special calls to the kernel debugger.

F.5.2.3 System Service Call

Format:

callsys

! PALcode format

Operation:

```
{ v0 = System service code }
{ a0-a5 = System call arguments }
previousPSR ← PSR
if( PSR<MODE> EQ UserMode ) then
    PSR<MODE> ← KernelMode
    tp ← (IKSP - TrapFrameLength) ! Establish trap pointer
else
    tp ← (sp - TrapFrameLength) ! Establish trap pointer
endif
TrIntSp(tp) ← sp
TrIntFp(tp) ← fp
TrIntRa(tp) ← ra
TrIntGp(tp) ← gp
TrFir(tp) ← ra
TrPsr(tp) ← previousPSR
gp ← KGP
sp ← tp
fp ← tp
t0 ← previousPSR<MODE>
t1 ← THREAD
RESTART_ADDRESS ← SYSCALL_ENTRY
```

GPR State Change:

```
fp ← tp
gp ← KGP
sp ← tp
t0 ← PSR
t1 ← THREAD
at and t0–t12 are UNPREDICTABLE
```

IPR State Change:

```
PSR<MODE> ← KernelMode
RESTART_ADDRESS ← SYSCALL_ENTRY
```

Exceptions:

```
Machine Checks
Kernel Stack Invalid
```

Description:

The `callsys` instruction raises a system service call exception to the kernel. The system service call has the software semantics of a standard procedure call. That is, arguments are passed in argument registers and on the stack, volatile registers are considered free, and nonvolatile registers must be preserved across the call. In addition to the standard calling sequence, `callsys` is passed the number of the desired system service in the return value register `v0`. `callsys` does not interpret this value, but rather passes it directly to the operating system.

`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. See Section F.4.1.6.

The argument registers must be preserved through the instruction. Standard control information, such as the previous PSR, is stored in the trap frame. `callsys` then restarts execution at the kernel system service call exception entry, passing the previous mode as a parameter in the `t0` register, and the current thread as a parameter in the `t1` register.

F.5.2.4 Generate a Trap

Format:

gentrap

! PALcode format

Operation:

{ a0 = Trap number that identifies exception }

See Sections F.4.1.7.8 and F.4.1.7.5

GPR State Change:

See Sections F.4.1.7.8 and F.4.1.7.5

IPR State Change:

See Sections F.4.1.7.8 and F.4.1.7.5

Exceptions:

Machine Checks

Kernel Stack Invalid

Description:

The gentrap instruction generates a software general exception 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.

F.5.2.5 Instruction Memory Barrier

Format:

imb

! PALcode format

Operation:

```
{ From within kernel mode, make processor }  
{ instruction stream coherent with main memory }
```

GPR State Change:

None

IPR State Change:

None

Exceptions:

Machine Checks

Description:

The imb instruction may only be called from kernel mode and guarantees that all subsequent instruction stream fetches are coherent with respect to main memory on the current processor. 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). See Section 6.7.3.

User-mode software must not use the imb instruction, but rather use the appropriate Windows NT interface to make the I-cache coherent.

F.5.2.6 Kernel Breakpoint Trap

Format:

kbpt

! PALcode format

Operation:

See Sections F.4.1.7.8 and F.4.1.7.6

GPR State Change:

See Sections F.4.1.7.8 and F.4.1.7.6

IPR State Change:

See Sections F.4.1.7.8 and F.4.1.7.6

Exceptions:

Machine Checks

Kernel Stack Invalid

Description:

The kbpt instruction raises a breakpoint general exception to the kernel, setting a KERNEL_BREAKPOINT breakpoint type.

F.5.2.7 Read Thread Environment Block Pointer

Format:

rdteb

! PALcode format

Operation:

v0 ← TEB

GPR State Change:

v0 ← TEB

IPR State Change:

None

Exceptions:

Machine Checks

Description:

The rdteb instruction returns in v0 the contents of the TEB internal processor register for the currently executing thread (the base address of the thread environment block). See Section F.2.7.

F.5.3 Debug PALcode and Free PALcode

The debug PALcode is a functional superset of the production PALcode, which is specified in this document. The debug PALcode includes extra counters for performance evaluation and additional sanity checks. An unacceptable performance loss would occur if these features were implemented in production PALcode. Therefore, the debug PALcode is used in the laboratory only.

The debug PALcode contains the following additional features:

- Kernel stack underflow/overflow checking
- Special I/O address checking
- Event counters

F.5.3.1 Kernel Stack Checking

The debug PALcode checks for kernel stack underflow and overflow whenever it allocates a trap frame and the previous mode was kernel mode. Two pages of kernel stack are allocated for each thread.

- Underflow occurs when the thread's kernel mode stack pointer (SP) is greater than the initial kernel stack pointer (IKSP).

- Overflow is detected whenever the SP would be less than $(IKSP - 2 * PAGE_SIZE)$.

Kernel stack underflow and overflow are indicated with a panic exception, described in Section F.4.1.8.

Implementation Note:

Alpha implementations that do not include the BWX extension (described in Appendix D) cannot provide direct access to I/O space addresses (as would Intel-based systems). Instead, those Alpha implementations provide access to I/O space by allowing the standard device drivers to use address handles, provided by the HAL, that may be treated as standard I/O virtual addresses for all operations except the I/O accesses. The I/O accesses must be performed by specialized routines in the HAL that are able to convert the address handles to the actual virtual addresses used for the I/O space accesses.

By convention, the HAL uses the range of numbers $A0000000_{16}$ through $BFFFFFFF_{16}$ to represent these address handles whenever possible. This range of numbers falls into the upper half of the 32-bit superpage address range. The debug PALcode disables the 32-bit superpage in hardware and provides support for the lower half of the 32-bit superpage in PALcode (the range of addresses 80000000_{16} through $9FFFFFFF_{16}$). Addresses in the range $A0000000_{16}$ through $BFFFFFFF_{16}$ are treated as standard addresses and, since they are not mapped, cause memory management faults (translation not valid). This support in the PALcode allows easy and precise trapping of device driver code that attempts to access I/O addresses directly, without using the intended access routines provided by the HAL.

Note:

Physical system memory is limited to 512M bytes when running with the debug PALcode.

F.5.3.2 Event Counters

The debug PALcode provides software counters to count significant events within the PALcode. The PALcode also provides the privileged `rdcounters` instruction to allow kernel-mode code to read the counters. The counted events are implementation specific but must include the following: a separate counter for each of the different PALcode instructions, TB miss counts, and interrupt counts. The format of the data returned by `rdcounters` is also implementation specific. However, all counters must be 64-bit counters.

F.6 Initialization and Firmware Transitions

This section describes the four phases of PALcode environment initialization and the PALcode functions that provide the transition between the operating system and the firmware.

F.6.1 Initialization

From the perspective of the PALcode environment there are four phases of initialization:

1. Internal system-specific processor state is established before the PALcode runs.
2. PALcode initializes the internal processor state.
3. The kernel uses PALcode initialization callback instructions to prepare the PALcode to handle exceptions.

4. Interrupt tables are initialized so that standard interrupt support can be used.

F.6.1.1 Pre-PALcode Initialization

Firmware must set the processor and system to a known good state before the PALcode entry point is called. The firmware must initialize any internal processor registers that contain system-specific parameters such as timing or memory size information. This is necessary because the PALcode is entirely independent of the system. The firmware must ensure that all caches are coherent with main memory before calling the PALcode and that the memory system has been fully initialized.

Hardware Implementation Note:

If system configuration information is written to write-only IPRs, those configuration IPRs cannot have any control bits that need to be written by the platform-independent operating system PALcode. If such bits were written in that manner, the firmware would have to pass the configuration information in internal processor state on a per-implementation basis. Hardware designers should consider allowing configuration registers to be read as well as written to allow the platform-independent layer to have visibility to the full internal processor state.

F.6.1.2 PALcode Initialization

The PALcode is entered at the first instruction at the base of the PALcode image. PALcode is called with the page frame number (PFN) of the PCR as a parameter in a1. All other argument registers must be preserved across PALcode initialization and are considered parameters to the operating system and are not interpreted by the PALcode. That is, the PALcode is free to destroy volatile general-purpose integer and floating-point registers, but must preserve the non-volatile register state across the call. Register volatility is listed in Section F.1.2. The PALcode must accomplish the following initialization:

1. Deassert all interrupt requests and disable all interrupt enables (this includes software, hardware and asynchronous trap interrupts).
2. Set the processor status register (PSR) such that interrupts are enabled, interrupt request level is set to high level (7), and the mode is kernel.
3. Invalidate all virtual translation buffers.
4. Establish all required superpage mapping: 32-bit I-stream and D-stream, and 43-bit D-stream mapping.
5. Set the previous_PAL_BASE register to the previous value of the PAL_BASE register.
6. Set the PAL_BASE register to the base address of the PALcode image.
7. Set the interrupt level table so that no interrupts are enabled for all interrupt levels.
8. Initialize all architected internal processor registers to their specified initialization values.
9. Begin any required implementation-specific initialization, such as unlocking error registers.

When the PALcode has completed its initialization, it resumes execution at the address passed in the ra (return address) register.

F.6.1.3 Kernel Callback Initialization of PALcode

The kernel uses the `initpal` and `wrentry` instructions to call back into the PALcode with the initialization values that allow exceptions to be handled properly between the PALcode and the kernel.

The kernel uses `initpal` to establish system-permanent context and per-thread context for the initialization thread. The system-permanent context passed to `initpal` is the kernel global pointer (KGP), which is passed via the `gp` register.

The initialization thread data passed in `initpal` are the page directory page, the initial kernel stack pointer, and the initialization thread address. The page directory page and thread address are passed as standard parameters; the kernel stack pointer is passed in the `sp` register. The `initpal` instruction also initializes the PALcode information section of the processor control region.

The kernel uses `wrentry` to register the kernel exception entry points with the PALcode. The `wrentry` instruction is called once for each kernel exception entry point. Each call includes the exception entry point address and the number of the exception class it handles.

F.6.1.4 Interrupt Table Initialization

The interrupt table values in the processor control region are system specific and so are not initialized until HAL initialization. Until these tables are initialized, the PALcode uses interrupt tables that are initialized such that all interrupts are disabled. An implementation may choose to cache some portion of the interrupt tables within the processor. After the operating system has established the interrupt tables, an implementation may use the `initpcr` instruction to cache some part of those tables.

F.6.2 Firmware Interfaces

The firmware PALcode environment is decoupled from the operating system PALcode. The `reboot/restart` and `swppal` instructions permit the transition between the operating system and the firmware PALcode context.

F.6.2.1 Reboot Instruction – Transition to Firmware PALcode Context

The `reboot` instruction performs a controlled transition to the firmware PALcode context. `Reboot` essentially follows the semantics for a return to the ARC (Advanced RISC Computing) firmware environment, with the addition of Alpha support for switching to the firmware PALcode. The `reboot` function accomplishes the following tasks:

1. Retrieves the restart block pointer from the processor control region.

The restart block is expected to be initialized by the firmware. The pointer to the restart block is passed by the firmware through the OS Loader to the kernel in the loader parameter block. The kernel writes the restart block pointer into the processor control region during startup. The restart block pointer must be a 32-bit superpage address.

The firmware environment is responsible for allocating memory for the entire restart block, including the saved state area that is specific to the Alpha architecture. The firmware is also responsible for initializing the restart block, as specified by ARC.

2. Verifies the restart block and if invalid, initiates alternate restart.

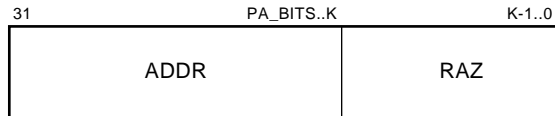
The PALcode verifies the restart block by ensuring that the restart block signature is

valid and that the restart block and saved state area lengths are of sufficient size to contain the state the PALcode saves. If the PALcode determines that the restart block is not valid, an alternate restart is initiated.

The alternate restart allows the PALcode to restore the previous PALcode base to the PAL_BASE register and to transfer control to the previous PALcode base in the PALcode environment.

Figure F-9 shows the structure of the PAL_BASE register.

Figure F-9: PAL_BASE Internal Processor Register



The hardware vectors into the appropriate PALcode handlers as offsets from the base in the PAL_BASE register. The offsets for each handler and the type of handler are implementation specific, except for the reset vector. The reset vector is the PALcode initialization vector and must begin at offset 0 within the PALcode image.

Explicitly, PAL_BASE contains the value $\langle PA_BITS..K \rangle$, where PA_BITS is the physical address bits for the implementation, and $2^{**}K$ is the minimum PALcode byte alignment for the implementation.

Note that the OS Loader uses 64K-byte boundaries, so the maximum value for K is 16. The minimum value for K is N , where $2^{**}N = \text{implementation page size}$.

3. Saves the general register state in the restart block.

The saved general register state includes all 32 integer registers and all 32 floating-point registers. In addition, the floating-point control register is also saved.

4. Saves the architected internal processor register state in the restart block.

The internal processor register state is stored in its architected format so that it may be interpreted in the firmware environment. In addition, remaining space is allocated so that the total size of the restart block is 2040 bytes. The additional space can be used for per-implementation data.

5. Saves the RESTART_ADDRESS in the restart block.

The RESTART_ADDRESS is stored in the saved state area to allow return from reboot via the restart instruction. The HAL is responsible for populating the Version, Revision, and RestartAddress fields of the restart block header.

6. Retrieves the firmware restart address from the processor control region.

The firmware restart address is the address to which the PALcode transfers control upon completion of the reboot. The firmware restart address is passed from the firmware through the OS Loader to the kernel and stored in the processor control region as is the restart block pointer. The firmware restart address is read from the processor control region and written to the RESTART_ADDRESS register with implementation-specific (but well-defined) interpretation.

7. Restores the PALcode base from the previous PALcode base.

The PALcode captures the previous PALcode environment when it is first initialized.

The PALcode base address is read from the PAL_BASE register and written to the previous_PAL_BASE register. When the processor executes the reboot function, it restores the previous PALcode environment by writing the value in the previous_PAL_BASE register to the PAL_BASE register.

Hardware Implementation Note:

Several restrictions are imposed on the hardware design to support this model for switching PALcode environments:

- A. The currently active PALcode must be settable by writing the base address of the PALcode image to an internal processor register.
 - B. No implementation can require, for the base of the PALcode, an alignment of greater than 64K bytes or less than the implementation page size.
 - C. The internal processor register used to set the base of the PALcode must be readable for each bit that is writeable.
8. Completes the restart block by updating the boot status and the checksum.
 9. Restarts execution at the firmware restart address passing a pointer to the restart block in the a0 register.

The restart instruction is provided to reverse the work done by a reboot instruction and allows the processor to restart execution. The restart function performs the inverse of the tasks that were performed in the reboot.

F.6.2.2 Reboot and Restart Tasks and Sequence

The tasks and sequence required for performing a reboot and restart are described below:

1. Firmware allocates restart block, initializing signature, length, ID fields, and the pointer to next restart block. Restart block pointer and firmware restart address are passed to the kernel.
2. HAL populates the Version and Revision fields during HAL initialization.
3. Some external event triggers a halt, a reboot, or a power-fail.
4. The appropriate HAL routine populates the RestartAddress field of the restart block with the address of the HAL restart routine.
5. The HAL executes the reboot instruction.
6. The PALcode saves processor state, including the RESTART_ADDRESS register (the address in the HAL of the instruction after the reboot instruction).
7. The PALcode transfers to the firmware environment.
8. The firmware initializes a restart by calling the HAL restart routine (via the address in the restart block header).
9. The HAL uses the swppal instruction to restore the operating system PALcode environment.
10. The HAL uses the restart instruction to restore complete processor state.
11. The PALcode restores state and then returns execution to the instruction after the reboot instruction in the HAL.
12. The HAL completes the restart.

F.6.2.3 Swppal Instruction – Transition to Any PALcode Environment

The swppal instruction is a flexible interface that allows kernel code to transition to any PALcode environment, as contrasted with reboot, which limits the caller to transition to the previous PALcode environment.

F.7 Windows NT Alpha Instruction Summary

Table F–24: Windows NT Alpha Unprivileged PALcode Instructions

Mnemonic	Opcode	Description
bpt	00.0080	Breakpoint trap
callkd	00.00AD	Call kernel debugger
callsys	00.0083	Call system service
gentrap	00.00AA	Generate trap
imb	00.0086	Instruction memory barrier
kbpt	00.00AC	Kernel breakpoint trap
rdteb	00.00AB	Read TEB internal processor register

Table F–25: Windows NT Alpha Privileged PALcode Instructions

Mnemonic	Opcode	Description
csir	00.000D	Clear software interrupt request
di	00.0008	Disable interrupts
draina	00.0002	Drain aborts
dtbis	00.0016	Data translation buffer invalidate single
ei	00.0009	Enable interrupts
halt	00.0000	Trap to illegal instruction
initpal	00.0004	Initialize the PALcode
rdcounters	00.0030	Read PALcode event counters
rdirql	00.0007	Read current IRQL
rdksp	00.0018	Read initial kernel stack
rdmces	00.0012	Read machine check error summary
rdpcr	00.001C	Read PCR (processor control registers)
rdpsr	00.001A	Read processor status register
rdstate	00.0031	Read internal processor state
rdthread	00.001E	Read the current thread value
reboot	00.0002	Transfer to console firmware
restart	00.0001	Restart the processor
retsys	00.000F	Return from system service call
rfe	00.000E	Return from exception
ssir	00.000C	Set software interrupt request
swpctx	00.0010	Swap privileged thread context

Table F–25: Windows NT Alpha Privileged PALcode Instructions (Continued)

Mnemonic	Opcode	Description
swpirql	00.0006	Swap IRQL
swpksp	00.0019	Swap initial kernel stack
swppal	00.000A	Swap PALcode
swpprocess	00.0011	Swap privileged process context
tbia	00.0014	Translation buffer invalidate all
tbis	00.0015	Translation buffer invalidate single
tbisasn	00.0017	Translation buffer invalidate single ASN
wrentry	00.0005	Write system entry
wrmces	00.0013	Write machine check error summary
wrperfmon	00.0020	Write performance monitoring values

Opcodes 00.003816 through 00.003F16 are reserved for processor implementation-specific PALcode instructions. All other opcodes are reserved for use by Compaq.

Index Entries are keyed with the following suffixes:

Suffix	Location
(I)	Common architecture
(II-A)	OpenVMS
(II-B)	Tru64 UNIX
(II-C)	Alpha Linux
(III)	Console interface
None	The specified appendix.

A

- Aborts, forcing, 6–5 (I)
- Absolute longword queue, 10–20 (II-A)
- Absolute quadword queue, 10–23 (II-A)
- Access control violation (ACV) fault, 11–15 (II-A), 14–10 (II-A)
 - memory protection, 11–8 (II-A)
 - precedence, 11–16 (II-A)
 - service routine entry point, 14–25 (II-A)
- Access violation fault, 17–13 (II-B), 22–13 (II-C), F–22
- ACCESS(x,y) operator, 3–6 (I)
- Add instructions
 - add longword, 4–26 (I)
 - add quadword, 4–28 (I)
 - add scaled longword, 4–27 (I)
 - add scaled quadword, 4–29 (I)
 - See also Floating-point operate
- ADDF instruction, 4–109 (I)
- ADDG instruction, 4–109 (I)
- ADDL instruction, 4–26 (I)
- ADDQ instruction, 4–28 (I)
- Address space, F–13
- Address space match (ASM)
 - bit in PTE, 11–5 (II-A), 17–5 (II-B), 22–5 (II-C), F–16
 - context switching and, F–12, F–71
 - TBIAP register uses, 13–26 (II-A)
 - virtual cache coherency, 5–4 (I)
- Address space number (ASN) register, 13–4 (II-A), F–7
 - context switching and, F–12
 - defined, 15–2 (II-B), 20–2 (II-C)
 - described, 17–12 (II-B), 22–12 (II-C)
 - HWPCB and, 12–2 (II-A)
 - HWPCB, initial and, 27–25 (III)
 - PALcode switching and, 27–8 (III)
 - privileged context, 10–90 (II-A)
 - process context and, 18–1 (II-B), 23–1 (II-C)
 - processor initialization and, 27–23 (III)
 - range supported, 11–14 (II-A)
 - TBCHK register uses, 13–24 (II-A)
 - TBIS register uses, 13–27 (II-A)
 - translation buffers and, 11–13 (II-A)
 - virtual cache coherency, 5–4 (I)
- Address translation
 - algorithm to perform, 11–9 (II-A), 11–12 (II-A), 17–11 (II-B), 22–11 (II-C)
 - page frame number (PFN), 11–8 (II-A)
 - page table structure, 11–8 (II-A), F–14
 - performance enhancements, 11–10 (II-A), 11–13 (II-A), 17–12 (II-B), 22–12 (II-C), E–6
 - physical, 17–7 (II-B), 22–7 (II-C)
 - translation buffers and, 11–13 (II-A)

- virtual, 17–9 (II-B), 22–9 (II-C)
- virtual address segment fields, 11–8 (II-A)
- ADDS instruction, 4–110 (I)
- ADDI instruction, 4–110 (I)
- AFTER, defined for memory access, 5–13 (I)
- Aligned byte/word memory accesses, A–11
- ALIGNED data objects, 1–8 (I)
- Alignment
 - atomic byte, 5–3 (I)
 - atomic longword, 5–2 (I)
 - atomic quadword, 5–2 (I)
 - D_floating, 2–5 (I)
 - data alignment trap, 14–14 (II-A)
 - data considerations, A–5
 - double-width data paths, A–1
 - F_floating, 2–4 (I)
 - G_floating, 2–5 (I)
 - instruction, A–2
 - longword, 2–2 (I)
 - longword integer, 2–11 (I)
 - memory accesses, A–11
 - program counter (PC), 14–6 (II-A)
 - quadword, 2–3 (I)
 - quadword integer, 2–11 (I)
 - S_floating, 2–8 (I)
 - stack, 14–29 (II-A)
 - T_floating, 2–9 (I)
 - unaligned data and, 14–26 (II-A)
 - X_floating, 2–9 (I)
- Alpha architecture
 - addressing, 2–1 (I)
 - overview, 1–1 (I)
 - porting operating systems to, 1–1 (I)
 - programming implications, 5–1 (I)
 - registers, 3–1 (I)
 - security, 1–6 (I)
 - See also Conventions
- Alpha finite number, 4–64 (I)
- Alpha Linux PALcode, instruction summary, C–20
- Alpha privileged architecture library. See PALcode
- AMASK (architecture mask) instruction, 4–133 (I)
 - arithmetic trap completion and, 4–73 (I)
 - bit assignments, D–4
- AMOVRM (PALcode) instruction, 10–74 (II-A)
- AMOVR (PALcode) instruction, 10–74 (II-A)
- AND instruction, 4–43 (I)
- AND operator, 3–7 (I)
- APC_LEVEL, IRQL table index name, F–5
- ARC Restart Block, F–59
- Architecture extensions, AMASK and, 4–133 (I)
- ARITH_RIGHT_SHIFT(x,y) operator, 3–7 (I)
- Arithmetic exceptions, F–23
 - See also Arithmetic traps
- Arithmetic instructions, 4–25 (I)
 - See also specific arithmetic instructions
- Arithmetic left shift instruction, 4–42 (I)
- Arithmetic trap completion, 4–73 (I)
 - AMASK instruction and, 4–73 (I)
- Arithmetic trap entry (entArith) register, 15–2
 - (II-B), 19–4 (II-B), 20–2 (II-C), 24–4 (II-C)
- Arithmetic traps, F–23
 - completion, 4–73 (I)
 - concurrent with data alignment, 14–14 (II-A)
 - denormal operand exception disabling, 4–82 (I)
 - denormal operand exception enabling, B–5
 - denormal operand status of, B–5
 - described, 14–11 (II-A)
 - disabling, 4–79 (I)
 - division by zero, 4–78 (I), 4–81 (I), 14–13
 - (II-A), 19–6 (II-B), 24–6 (II-C), F–25
 - division by zero, disabling, 4–82 (I)
 - division by zero, enabling, B–6
 - division by zero, status of, B–5
 - dynamic rounding mode, 4–81 (I)
 - enabling, B–4
 - F31 as destination, 14–11 (II-A)
 - inexact result, 4–78 (I), 4–81 (I), 14–13 (II-A),
 - 19–5 (II-B), 24–5 (II-C), F–25
 - inexact result, disabling, 4–81 (I)
 - inexact result, enabling, B–5
 - inexact result, status of, B–5
 - integer overflow, 4–79 (I), 4–81 (I), 14–14
 - (II-A), 19–5 (II-B), 24–5 (II-C), F–25
 - integer overflow, disabling, B–4
 - integer overflow, enabling, B–4
 - invalid operation, 4–77 (I), 4–81 (I), 14–13
 - (II-A), 19–6 (II-B), 24–6 (II-C), F–25
 - invalid operation, disabling, 4–82 (I)
 - invalid operation, enabling, B–6
 - invalid operation, status of, B–5
 - overflow, 4–78 (I), 4–81 (I), 14–13 (II-A),
 - 19–5 (II-B), 24–5 (II-C), F–25
 - overflow, disabling, 4–82 (I)
 - overflow, enabling, B–6
 - overflow, status of, B–5
 - program counter (PC) value, 14–13 (II-A)
 - programming implications for, 5–29 (I)

- R31 as destination, 14–11 (II-A)
 - recorded for software, 14–12 (II-A)
 - registers, when affected by, 14–13 (II-A)
 - REI instruction with, 14–9 (II-A)
 - service routine entry point, 14–25 (II-A)
 - system entry for, 19–4 (II-B), 24–4 (II-C)
 - TRAPB instruction with, 4–147 (I)
 - underflow, 4–81 (I), 14–13 (II-A), 19–5 (II-B), 24–5 (II-C), F–25
 - underflow, enabling, B–5
 - underflow, status of, B–5
 - ASCII character set, C–26
 - ASN_wrap_indicator, F–12
 - AST enable (ASTEN) register
 - changing access modes in, 12–4 (II-A)
 - described, 13–5 (II-A)
 - HWPCB and, 12–2 (II-A)
 - HWPCB, initial and, 27–25 (III)
 - in initial HWPCB, 27–25 (III)
 - interrupt arbitration, 14–32 (II-A)
 - operation (with ASTs), 12–4 (II-A)
 - privileged context, 10–90 (II-A)
 - processor initialization and, 27–23 (III)
 - SWASTEN instruction with, 10–18 (II-A)
 - AST summary (ASTSR) register
 - described, 13–7 (II-A)
 - HWPCB and, 12–2 (II-A)
 - HWPCB, initial and, 27–25 (III)
 - indicates pending ASTs, 12–4 (II-A)
 - interrupt arbitration, 14–32 (II-A)
 - privileged context, 10–90 (II-A)
 - processor initialization and, 27–23 (III)
 - Asynchronous procedure call (APC)
 - SIRR register field for, F–34
 - Asynchronous system traps (AST)
 - ASTEN/ASTSR registers with, 12–3 (II-A)
 - initiating, 12–4 (II-A)
 - interrupt, defined, 14–18 (II-A)
 - PS register and, 12–4 (II-A)
 - service routine entry point, 14–26 (II-A)
 - Atomic access, 5–3 (I)
 - Atomic move operations, 10–73 (II-A)
 - Atomic operations
 - load-locked and store conditional, using, 5–7 (I)
 - longword datum, accessing, 5–2 (I)
 - low-contention prefetching, 5–8 (I)
 - page table entry, modifying, 11–6 (II-A)
 - quadword datum, accessing, 5–2 (I)
 - shared data structures, updating, 5–7 (I)
 - Atomic sequences, A–17
 - AUTO_ACTION environment variable, 26–26 (III)
 - cold bootstrap, 27–9 (III)
 - error halts, 27–34 (III)
 - overriding, 27–31 (III)
 - state transitions and, 27–1 (III)
 - system restarts, 27–32 (III)
- ## B
-
- BB_WATCH
 - powerfail interrupts, 27–32 (III)
 - power-up initialization, 27–4 (III)
 - primary console switching, 27–35 (III)
 - primary-eligible (PE) bit and, 27–47 (III)
 - requirements, 27–46 (III)
 - BEFORE, defined for memory access, 5–13 (I)
 - BEQ instruction, 4–21 (I)
 - BGE instruction, 4–21 (I)
 - BGT instruction, 4–21 (I)
 - BIC instruction, 4–43 (I)
 - Big-endian addressing, 2–12 (I)
 - byte operation examples, 4–55 (I)
 - byte swapping for, A–13
 - extract byte with, 4–52 (I)
 - insert byte with, 4–56 (I)
 - load F_floating with, 4–91 (I)
 - load long/quad locked with, 4–9 (I)
 - load S_floating with, 4–93 (I)
 - mask byte with, 4–58 (I)
 - store byte/word with, 4–16 (I)
 - store F_floating with, 4–95 (I)
 - store long/quad conditional with, 4–13 (I)
 - store long/quad with, 4–16 (I)
 - store S_floating with, 4–97 (I)
 - Big-endian data types, X_floating, 2–10 (I)
 - BIS instruction, 4–43 (I)
 - BITMAP_CHECKSUM
 - distributed memory cluster descriptor field, 27–16 (III)
 - static memory cluster descriptor field, 27–12 (III)
 - BITMAP_PA
 - distributed memory cluster descriptor field, 27–16 (III)
 - static memory cluster descriptor field, 27–12 (III)
 - BITMAP_VA
 - static memory cluster descriptor field, 27–12 (III)
 - BLBC instruction, 4–21 (I)

- BLBS instruction, 4–21 (I)
 - BLE instruction, 4–21 (I)
 - BLT instruction, 4–21 (I)
 - BNE instruction, 4–21 (I)
 - Boolean instructions, 4–42 (I)
 - logical functions, 4–43 (I)
 - Boolean stylized code forms, A–15
 - Boot block on disk, 27–40 (III)
 - Boot environment, restoring, F–59
 - Boot sequence, establishing, F–2
 - BOOT_DEV environment variable, 26–26 (III)
 - loading system software and, 27–22 (III)
 - BOOT_FILE environment variable, 26–26 (III), 27–42 (III)
 - loading system software and, 27–23 (III)
 - BOOT_OSFLAGS environment variable, 26–27 (III)
 - loading system software and, 27–23 (III)
 - BOOT_RESET environment variable, 26–27 (III)
 - cold bootstrap, 27–9 (III)
 - overriding, 27–31 (III)
 - system initialization, 27–3 (III)
 - warm bootstrap, 27–25 (III)
 - BOOTDEF_DEV environment variable, 26–26 (III)
 - loading system software and, 27–22 (III)
 - BOOTED_DEV environment variable
 - loading system software and, 27–22 (III)
 - BOOTED_FILE environment variable, 26–27 (III)
 - loading system software and, 27–23 (III)
 - BOOTED_OSFLAGS environment variable, 26–27 (III)
 - loading system software and, 27–23 (III)
 - BOOTP-UDP/IP network protocol, 27–45 (III)
 - Bootstrap address space
 - regions, 27–17 (III)
 - Bootstrap-in-progress (BIP) flag
 - failed bootstrap and, 27–21 (III)
 - multiprocessor booting and, 27–27 (III)
 - per-CPU state contains, 26–23 (III)
 - power-up initialization and, 27–4 (III)
 - processor initialization and, 27–23 (III)
 - secondary console and, 27–30 (III)
 - state transitions and, 27–1 (III)
 - Bootstrapping, 27–1 (III)
 - adding processor while running system, 27–30 (III)
 - address space at cold, 27–17 (III)
 - boot block in ROM, 27–44 (III)
 - boot block on disk, 27–40 (III)
 - cold in uniprocessor environment, 27–8 (III)
 - control to system software, 27–24 (III)
 - disk, from, 27–40 (III)
 - failure of, 27–21 (III)
 - implementation considerations, 27–47 (III)
 - loading page table space at cold, 27–18 (III)
 - loading primary image, 27–39 (III)
 - loading system software, 27–22 (III)
 - magtape, from, 27–42 (III)
 - MOP-based network, from, 27–45 (III)
 - multiprocessor, 27–27 (III)
 - PALcode loading at cold, 27–17 (III)
 - processor initialization, 27–23 (III)
 - request from system software, 27–31 (III)
 - ROM, from, 27–44 (III)
 - state flags with, 27–21 (III)
 - system, 27–3 (III)
 - unconditional, 27–31 (III)
 - warm, 27–25 (III)
- BPT (PALcode) instruction, 10–4 (II-A)
 - required recognition of, 6–4 (I)
 - service routine entry point, 14–26 (II-A)
 - trap information, 14–15 (II-A)
 - bpt (PALcode) instruction, 16–2 (II-B), 21–2 (II-C), F–82
 - required recognition of, 6–4 (I)
 - BR instruction, 4–22 (I)
 - lock_flag with, 4–10 (I)
 - Branch instructions, 4–19 (I)
 - backward conditional, 4–21 (I)
 - conditional branch, 4–21 (I)
 - floating-point, summarized, 4–99 (I)
 - format of, 3–11 (I)
 - forward conditional, 4–21 (I)
 - opcodes and format summarized, C–1
 - unconditional branch, 4–22 (I)
 - See also Control instructions
 - Branch prediction model, 4–19 (I)
 - Branch prediction stack, with BSR instruction, 4–22 (I)
 - Breakpoint exceptions, F–27
 - initiating, 10–4 (II-A)
 - Breakpoint trap, initiating, 16–2 (II-B), 21–2 (II-C)
 - BSR instruction, 4–22 (I)
 - lock_flag with, 4–10 (I)
 - Bugcheck exception, initiating, 10–5 (II-A)

BUGCHK (PALcode) instruction, 10–5 (II-A)
 required recognition of, 6–4 (I)
 service routine entry point, 14–26 (II-A)
 trap information, 14–15 (II-A)

bugchk (PALcode) instruction, 16–3 (II-B), 21–3
 (II-C)
 required recognition of, 6–4 (I)

Byte data type, 2–1 (I)
 atomic access of, 5–3 (I)

Byte manipulation, 1–2 (I)

Byte manipulation instructions, 4–48 (I)

Byte swapping, A–13

Byte_within_page field, 11–2 (II-A), 17–2 (II-B),
 22–2 (II-C)

BYTE_ZAP(x,y) operator, 3–7 (I)

C

/C qualifier
 IEEE chopped rounding, 4–68 (I)
 VAX chopped rounding, 4–68 (I)

Cache blocks, virtual
 invalidating all, F–72
 invalidating multiple, F–73
 invalidating single, F–75

Cache coherency, F–10
 barrier instructions for, 5–25 (I)
 defined, 5–2 (I)
 HAL interface for, F–3
 multiprocessor environment and, 5–6 (I)

Caches
 design considerations, A–2
 flushing physical page from, 10–82 (II-A),
 16–11 (II-B), 21–10 (II-C)
 I-stream considerations, A–5
 MB and IMB instructions with, 5–25 (I)
 powerfail/recovery and, 5–5 (I)
 requirements for, 5–4 (I)
 translation buffer conflicts, A–7

CALL_PAL (call privileged architecture library)
 instruction, 4–135 (I)

CALL_PAL instruction
 lock_flag with, 4–10 (I)

callkd (PALcode) instruction, F–83

callsys (PALcode) instruction, 16–4 (II-B), 21–4
 (II-C), F–84
 entSys with, 19–9 (II-B), 24–9 (II-C)
 stack frames for, 19–3 (II-B), 24–3 (II-C)

CASE operator, 3–7 (I)

Catastrophic errors, F–37

Causal loops, 5–15 (I)

CFLUSH (PALcode) instruction, 10–82 (II-A)
 ECB compared with, 4–138 (I)
 powerfail and, 14–20 (II-A)

cflush (PALcode) instruction, 16–11 (II-B), 21–10
 (II-C)

Changed datum, 5–6 (I)

CHAR_SET environment variable, 26–28 (III)

Characters
 getting from console, 26–34 (III)
 writing to console terminal, 26–36 (III)

Charged process cycles register, 10–90 (II-A)
 HWPCB and, 12–2 (II-A)
 PCC register and, 12–3 (II-A)
 process context and, 18–1 (II-B), 23–1 (II-C)

CHECKSUM
 distributed memory cluster descriptor field,
 27–15 (III)
 HWRPB field, 26–9 (III)
 memory data descriptor table field, 27–11 (III)
 multiprocessor boot and, 27–27 (III)
 null memory cluster descriptor field, 27–13 (III)

CHME (PALcode) instruction, 10–6 (II-A)
 service routine entry point, 14–26 (II-A)
 trap initiation, 14–16 (II-A)

CHMK (PALcode) instruction, 10–7 (II-A)
 service routine entry point, 14–26 (II-A)
 trap initiation, 14–16 (II-A)

CHMS (PALcode) instruction, 10–8 (II-A)
 service routine entry point, 14–26 (II-A)
 trap initiation, 14–16 (II-A)

CHMU (PALcode) instruction, 10–9 (II-A)
 service routine entry point, 14–26 (II-A)
 trap initiation, 14–16 (II-A)

Clear a register, A–14

Clock. See BB_WATCH

CLOCK_HIGH, IRQI table index name, F–5

CLOSE device routine, 26–50 (III)

CLRFEN (PALcode) instruction, 10–10 (II-A)

clrfen (PALcode) instruction, 16–5 (II-B), 21–5
 (II-C)

CLUSTER
 memory data descriptor table field, 27–12 (III)

CLUSTERS

- memory data descriptor table field, 27–12 (III)
- null memory cluster descriptor field, 27–13 (III)
- Clusters, memory, 27–9 (III)
- CMOVEQ instruction, 4–44 (I)
- CMOVGE instruction, 4–44 (I)
- CMOVGT instruction, 4–44 (I)
- CMOVLBC instruction, 4–44 (I)
- CMOVLE instruction, 4–44 (I)
- CMOVLTL instruction, 4–44 (I)
- CMOVNE instruction, 4–44 (I)
- CMPBGE instruction, 4–50 (I)
 - endian considerations with, 2–12 (I)
- CMPEQ instruction, 4–30 (I)
- CMPGEQ instruction, 4–111 (I)
- CMPGLE instruction, 4–111 (I)
- CMPGLT instruction, 4–111 (I)
- CMPL instruction, 4–30 (I)
- CMPLT instruction, 4–30 (I)
- CMPTEQ instruction, 4–112 (I)
- CMPTLE instruction, 4–112 (I)
- CMPTLT instruction, 4–112 (I)
- CMPTUN instruction, 4–112 (I)
- CMPULE instruction, 4–31 (I)
- CMPULT instruction, 4–31 (I)
- Code forms, stylized, A–13
 - boolean, A–15
 - clear register, A–14
 - load literal, A–14
 - negate, A–15
 - NOP, A–13
 - NOT, A–15
 - register-to-register move, A–15
- Code scheduling
 - IMPLVER instruction with, 4–141 (I)
- Code sequences, A–11
- CODEC, 4–154 (I)
- Coherency
 - cache, 5–2 (I)
 - memory, 5–1 (I)
- Compare instructions
 - compare integer signed, 4–30 (I)
 - compare integer unsigned, 4–31 (I)
 - See also Floating-point operate
- Conditional move instructions, 4–44 (I)
 - See also Floating-point operate
- CONFIG block, in HWRPB, 26–10 (III)
- CONFIG offset, HWRPB field for, 26–8 (III)
- CONFIG. See Configuration data block
- Configuration data block, 26–23 (III)
- Console
 - character sets, 26–29 (III)
 - closing terminal, 26–45 (III)
 - console I/O mode, 27–3 (III)
 - data log length, 26–21 (III)
 - data log physical address, 26–21 (III)
 - data structure linkage, 26–69 (III)
 - data structures loading at cold boot, 27–17 (III)
 - definition, 25–1 (III)
 - detached, 25–2 (III)
 - detached implementations of, 27–49 (III)
 - embedded, 25–2 (III)
 - embedded implementation of, 27–47 (III)
 - environment variables, required, 26–26 (III)
 - error halt and recovery, 27–34 (III)
 - forcing entry to I/O mode, 27–39 (III)
 - HWRPB with, 26–1 (III)
 - implementation registry, 25–3 (III)
 - implementations, 25–2 (III)
 - inter-console communications buffer, 26–77 (III)
 - internationalization, 25–4 (III)
 - interprocessor communications for, 26–75 (III)
 - ISO Latin-1 support with, 25–4 (III)
 - loading PALcode, 27–17 (III)
 - loading system software, 27–22 (III)
 - lock mechanisms, 25–2 (III)
 - major state transitions, 27–2 (III)
 - messages for, 25–4 (III)
 - miscellaneous routines, 26–63 (III)
 - multiprocessor boot, 27–27 (III)
 - multiprocessor implementation of, 27–48 (III)
 - opening terminal, 26–44 (III)
 - presentation layer, 25–3 (III)
 - processor state flags, 27–21 (III)
 - program I/O mode, 27–3 (III)
 - remapping routines, 26–71 (III)
 - requirements for, 25–2 (III)
 - resetting, 26–38 (III)
 - RESTORE_TERM routine, 27–37 (III), 27–39 (III)
 - SAVE_TERM routine, 27–37 (III), 27–38 (III)
 - secondary at multiprocessor boot, 27–29 (III)
 - security for, 25–4 (III)
 - sending commands to secondary, 26–77 (III)
 - sending messages to primary, 26–78 (III)

- switching primary processors, 26–63 (III)
- system restarts and, 27–31 (III)
- warm bootstrap and, 27–25 (III)
- Console callback routine block, in HWRPB, 26–10 (III)
- Console callback routines, 26–29 (III)
 - cold boot and, 27–17 (III)
 - CTB describes, 26–73 (III)
 - data structures for, 26–68 (III)
 - fixing up the virtual address, 26–64 (III)
 - HWRPB field for, 26–8 (III)
 - remapping, 26–71 (III)
 - summary of, 26–30 (III)
 - system software invoking, 26–30 (III)
- Console data log length, 26–21 (III)
- Console data log physical address, 26–21 (III)
- Console environment variables
 - loading system software and, 27–22 (III)
 - See also Environment variables
- Console firmware, transferring to, F–59
- Console I/O mode, 27–3 (III)
 - forcing entry to, 27–39 (III)
- Console initialization mode, 27–3 (III)
- Console interface, 26–1 (III)
- Console overview, 7–1 (I)
- Console routine block (CRB), 26–68 (III)
 - console callback routines with, 26–68 (III)
 - initializing, 26–71 (III)
 - offset, HWRPB field for, 26–8 (III)
 - structure of, 26–69 (III)
- Console terminal block (CTB)
 - console callback routines with, 26–68 (III)
 - described, 26–32 (III), 26–73 (III)
 - HWRPB fields for, 26–8 (III)
 - structure of, 26–74 (III)
- Console terminal routines, 26–32 (III)
- CONSOLE, system variation field, 26–13 (III)
- CONSOLE_CLOSE console terminal routine, 26–45 (III)
- CONSOLE_OPEN console terminal routine, 26–44 (III)
- Context switching
 - between address spaces, F–71
 - defined, 12–1 (II-A)
 - hardware, 12–1 (II-A)
 - initiating, 10–90 (II-A)
 - PDR register with, F–15
 - raising IPL while, 12–4 (II-A)
 - software, 12–2 (II-A)
 - thread, F–66
 - thread to process, F–12
 - thread to thread, F–11
 - See also Hardware
- Context valid (CV) flag
 - multiprocessor booting and, 27–27 (III)
 - per-CPU state contains, 26–22 (III)
 - processor initialization and, 27–23 (III)
- Control instructions, 4–19 (I)
- Conventions
 - code examples, 1–9 (I)
 - code flows, F–4
 - extents, 1–8 (I)
 - figures, 1–9 (I)
 - instruction format, 3–9 (I)
 - notation, 3–9 (I)
 - numbering, 1–6 (I)
 - ranges, 1–8 (I)
- Corrected error interrupts, logout area for, 14–23 (II-A)
- Count instructions
 - Count leading zero, 4–32 (I)
 - Count population, 4–33 (I)
 - Count trailing zero, 4–34 (I)
- CPU ID, HWRPB field for primary, 26–6 (III)
 - multiprocessor booting and, 27–27 (III)
- CPU slot offset, HWRPB field for, 26–7 (III)
- CPYS instruction, 4–104 (I)
- CPYSE instruction, 4–104 (I)
- CPYSN instruction, 4–104 (I)
- CSERVE (PALcode) instruction, 10–83 (II-A)
 - required recognition of, 6–4 (I)
- cserve (PALcode) instruction, 16–12 (II-B), 21–11 (II-C)
 - required recognition of, 6–4 (I)
- csir (PALcode) instruction, F–40
 - clears software interrupts, F–34
- CTB table, in HWRPB, 26–10 (III)
- CTB. See Console terminal block
- CTLZ instruction, 4–32 (I)
- CTPOP instruction, 4–33 (I)
- CTTZ instruction, 4–34 (I)
- Current mode field, in PS register, 14–6 (II-A)
- Current PALcode, 27–5 (III)

- Current PC, 14–2 (II-A)
- CVTDG instruction, 4–115 (I)
- CVTGD instruction, 4–115 (I)
- CVTGF instruction, 4–115 (I)
- CVTGQ instruction, 4–113 (I)
- CVTLQ instruction, 4–105 (I)
- CVTQF instruction, 4–114 (I)
- CVTQG instruction, 4–114 (I)
- CVTQL instruction, 4–105 (I)
 - FP_C quadword with, B–4
- CVTQS instruction, 4–117 (I)
- CVTQT instruction, 4–117 (I)
- CVTST instruction, 4–119 (I)
- CVTTQ instruction, 4–116 (I)
 - FP_C quadword with, B–4
- CVTTS instruction, 4–118 (I)
- Cycle counter frequency
 - HWRPB field for, 26–7 (III)
 - per-CPU slot field, 26–21 (III)

D

- /D qualifier
 - IEEE dynamic rounding, 4–68 (I)
- D_floating data type, 2–5 (I)
 - alignment of, 2–5 (I)
 - mapping, 2–5 (I)
 - restricted, 2–5 (I)
- dalnfix (PALcode) instruction, F–41, F–45
- Data alignment, A–5
- Data alignment trap (DAT) register
 - privileged context, 10–90 (II-A)
- Data alignment traps, 14–14 (II-A)
 - concurrent with arithmetic, 14–14 (II-A)
 - fixup (DAT) bit, in HWPCB, 12–2 (II-A)
 - fixup (DATFX) register, 13–9 (II-A)
 - registers used, 14–14 (II-A)
 - service routine entry point, 14–26 (II-A)
 - system entry for, 19–9 (II-B), 24–9 (II-C)
- Data caches
 - ECB instruction with, 4–136 (I)
 - WH64x instruction with, 4–148 (I)
- Data format, overview, 1–3 (I)
- Data sets, buffering large, 26–21 (III)

- Data sharing (multiprocessor), A–6
 - preatching with, 5–8 (I)
 - synchronization requirement, 5–6 (I)
- Data stream considerations, A–5
- Data stream translation buffer (DTB), 26–13 (III)
- Data structures, shared, 5–6 (I)
- Data types
 - byte, 2–1 (I)
 - IEEE floating-point, 2–6 (I)
 - longword, 2–2 (I)
 - longword integer, 2–10 (I)
 - quadword, 2–2 (I)
 - quadword integer, 2–11 (I)
 - unsupported in hardware, 2–11 (I)
 - VAX floating-point, 2–3 (I)
 - word, 2–1 (I)
- DATA_BUS_ERROR code, F–36
- Deferred procedure call (DPC)
 - SIRR register field for, F–34
 - stack for, F–11
- Denormal, 4–65 (I)
- Denormal operand exception disable, 4–82 (I)
- Denormal operand exception enable (DNOE)
 - FP_C quadword bit, B–5
- Denormal operand status (DNOS)
 - FP_C quadword bit, B–5
- Denormal operands to zero, 4–82 (I)
- Depends order (DP), 5–15 (I)
- Detached console, 25–2 (III)
- DEVICE ID, CTB field for, 26–74 (III)
- DEVICE TYPE, CTB field for, 26–74 (III)
- DEVICE_HIGH_LEVEL, IRQL table index name, F–5
- DEVICE_LEVEL, IRQL table index name, F–5
- Device-specific data (DSD), 26–75 (III)
- di (PALcode) instruction, F–42
- Dirty pages, tracking, F–17
- Dirty zero, 4–65 (I)
- Disk bootstrap image, 27–40 (III)
- DISPATCH console routine, 26–31 (III)
- DISPATCH procedure, 26–70 (III)
- DISPATCH, CRB fields for, 26–70 (III)
- DISPATCH_LEVEL, IRQL table index name, F–5

- Distributed memory cluster descriptor table, 27–12 (III)
 - fields, 27–15 (III)
 - format, 27–15 (III)
 - linking, 27–17 (III)
- DIV operator, 3–7 (I)
- DIVF instruction, 4–120 (I)
- DIVG instruction, 4–120 (I)
- Division
 - integer, A–12
 - performance impact of, A–12
- Division by zero bit, exception summary register, F–25
- Division by zero enable (DZEE)
 - FP_C quadword bit, B–6
- Division by zero status (DZES)
 - FP_C quadword bit, B–5
- Division by zero trap, 14–13 (II-A), 19–6 (II-B), 24–6 (II-C), F–25
- DIVS instruction, 4–121 (I)
- DIVT instruction, 4–121 (I)
- DMA control, HAL interface for, F–3
- DMK bit, machine check error summary register, F–36
- DNOD bit. See Denormal operand exception disable
- DNZ. See Denormal operands to zero
- DP. See Depends order
- DPC bit, machine check error summary register, 13–14 (II-A), 19–8 (II-B), 24–8 (II-C), F–36
- DRAINA (PALcode) instruction
 - required, 6–4 (I)
- draina (PALcode) instruction, F–43
 - machine checks and, F–36
 - required, 6–4 (I)
- DSC bit, machine check error summary register, 13–14 (II-A), 19–8 (II-B), 24–8 (II-C), F–36
- DSD LENGTH, CTB field for, 26–75 (III)
- DSD, CTB field for, 26–75 (III)
- DSRDB block, in HWRPB, 26–10 (III)
- DSRDB offset, HWRPB field for, 26–9 (III)
- DSRDB structure, 26–24 (III)
- DTB. See data stream translation buffer

- dtbis (PALcode) instruction, F–17, F–44
- DUMP_DEV environment variable, 26–27 (III)
- DYN bit. See Arithmetic traps, dynamic rounding mode
- Dynamic system recognition data block. See DSRDB
- DZE bit
 - exception summary parameter, 14–12 (II-A)
 - exception summary register, 19–6 (II-B), 24–6 (II-C), F–25
 - See also Arithmetic traps, division by zero
- DZED bit. See Trap disable bits, division by zero

E

- ealnfix (PALcode) instruction, F–45
- ECB (Evict data cache block) instruction, 4–136 (I)
 - CFLUSH (PALcode) instruction with, 4–138 (I)
- ei (PALcode) instruction, F–46
 - as synchronization function, F–33
- Embedded console, 25–2 (III)
- ENABLE_AUDIT environment variable, 26–27 (III), 27–40 (III)
- entArith. See Arithmetic trap entry
- entIF. See Instruction fault entry
- entInt. See Interrupt entry
- entMM. See Memory management fault entry
- ENTRY, CRB field for, 26–70 (III)
- entSys. See System call entry
- entUna. See Unaligned access fault
- Environment variables, 26–24 (III)
 - getting, 26–60 (III)
 - power-up initialization and, 27–4 (III)
 - processor initialization and, 27–23 (III)
 - resetting, 26–59 (III)
 - routines described, 26–57 (III)
 - saving, 26–61 (III)
 - setting, 26–58 (III)
- EQV instruction, 4–43 (I)
- Error halt and recovery, 27–34 (III)
- Error messages
 - console, 25–4 (III)
- Errors
 - correctable, F–35
 - correctable processor, 19–8 (II-B), 24–8 (II-C)
 - correctable system, 19–8 (II-B), 24–8 (II-C)
 - uncorrectable, F–35

EXCB (exception barrier) instruction, 4–138 (I),
 A–15
 FPCR and, 4–84 (I)

Exception classes, F–20
 registry of handling routines for, F–77
 values for, F–78

Exception dispatch, F–20

Exception handlers, B–2
 TRAPB instruction with, 4–147 (I)

Exception handling routines, registry for, F–77

Exception register write mask, 19–6 (II-B), 24–6
 (II-C)

Exception service routines
 entry point, 14–24 (II-A)
 introduced, 14–8 (II-A)

Exception summary parameter, 14–12 (II-A)

Exception summary register, 19–2 (II-B), 19–4
 (II-B), 24–2 (II-C), 24–4 (II-C), F–25

EXCEPTION_SUMMARY, F–25

Exceptional events, 14–1 (II-A)

ExceptionPC address, F–23

Exceptions
 actions, summarized, 14–2 (II-A)
 arithmetic, F–23
 breakpoint, F–27
 defined, 14–1 (II-A), 19–1 (II-B), 24–1 (II-C)
 F31 with, 3–2 (I)
 general class common dispatch, F–28
 general class of, F–23
 illegal instruction, F–25
 initializing entry points, F–92
 initiated before interrupts, 14–16 (II-A)
 initiated by PALcode, 14–29 (II-A)
 introduced, 14–8 (II-A)
 invalid address, F–26
 memory management class, F–22
 precise IEEE-format arithmetic, 4–69 (I)
 precise VAX-format arithmetic, 4–68 (I)
 processor state transitions, 14–34 (II-A)
 R31 with, 3–1 (I)
 returning from, F–21, F–63
 software, F–26
 stack frames for, 14–7 (II-A), 19–3 (II-B), 24–3
 (II-C)
 subsetted IEEE, F–27
 system service calls, F–23
 trap frames with, F–21
 trapping modes, 4–70 (I)

unaligned access, F–25
 See also Arithmetic traps

Executive read enable (ERE), bit in PTE, 11–4
 (II-A)

Executive stack pointer (ESP) register, 13–10 (II-A)
 HWPCB and, 12–2 (II-A)
 HWPCB, initial and, 27–25 (III)
 internal processor register, 13–1 (II-A)

Executive write enable (EWE), bit in PTE, 11–4
 (II-A)

EXTBL instruction, 4–52 (I)

Extended VA size, HWRPB field for, 26–6 (III)

EXTLH instruction, 4–52 (I)

EXTLL instruction, 4–52 (I)

EXTQH instruction, 4–52 (I)

EXTQL instruction, 4–52 (I)

Extract byte instructions, 4–52 (I)
 big-endian support with, 2–13 (I)

EXTWH instruction, 4–52 (I)

EXTWL instruction, 4–52 (I)

F

F_floating data type, 2–3 (I)
 alignment of, 2–4 (I)
 compared to IEEE S_floating, 2–7 (I)
 MAX/MIN, 4–66 (I)
 unaligned data and, 14–26 (II-A)

F31 as destination register, 3–2 (I)

Fault on execute (FOE), 11–15 (II-A), 14–11 (II-A),
 17–14 (II-B), 22–14 (II-C)
 bit in PTE, 11–5 (II-A), 17–5 (II-B), 22–5
 (II-C)
 service routine entry point, 14–25 (II-A)
 software usage of, 14–11 (II-A)

Fault on read (FOR), 11–15 (II-A), 14–10 (II-A),
 17–14 (II-B), 22–14 (II-C)
 bit in PTE, 11–6 (II-A), 17–6 (II-B), 22–6
 (II-C)
 service routine entry point, 14–25 (II-A)
 software usage of, 14–10 (II-A)

Fault on write (FOW), 11–15 (II-A), 14–10 (II-A),
 17–14 (II-B), 22–14 (II-C)
 bit in PTE, 11–6 (II-A), 17–6 (II-B), 22–6
 (II-C), F–17
 service routine entry point, 14–25 (II-A)
 software usage of, 14–11 (II-A)

- Faults, 14–9 (II-A), F–22
 - access control violation, 14–10 (II-A)
 - defined, 14–8 (II-A), 19–1 (II-B), 24–1 (II-C)
 - fault on execute, 14–11 (II-A), 17–13 (II-B), 22–13 (II-C)
 - fault on read, 14–10 (II-A), 17–13 (II-B), 22–13 (II-C)
 - fault on write, 14–10 (II-A), 17–13 (II-B), 22–13 (II-C)
 - floating-point disabled, 14–10 (II-A)
 - memory management, 17–13 (II-B), 22–13 (II-C)
 - MM flag, 14–9 (II-A)
 - program counter (PC) value, 14–8 (II-A)
 - REI instruction with, 14–8 (II-A)
 - translation not valid, 14–10 (II-A)
- FBEQ instruction, 4–100 (I)
- FBGE instruction, 4–100 (I)
- FBGT instruction, 4–100 (I)
- FBLE instruction, 4–100 (I)
- FBLT instruction, 4–100 (I)
- FBNE instruction, 4–100 (I)
- FCMOVEQ instruction, 4–106 (I)
- FCMOVGE instruction, 4–106 (I)
- FCMOVGT instruction, 4–106 (I)
- FCMOVLE instruction, 4–106 (I)
- FCMOVLT instruction, 4–106 (I)
- FCMOVNE instruction, 4–106 (I)
- FEN. See Floating-point enable
- FETCH (prefetch data) instruction, 4–139 (I)
- FETCH_M (prefetch data, modify intent) instruction, 4–139 (I)
- Field replaceable unit (FRU)
 - memory clusters with, 27–12 (III)
 - offset, HWRPB field for, 26–8 (III)
 - table description, 26–23 (III)
- Finite number, Alpha, contrasted with VAX, 4–64 (I)
- Firmware components, F–2
- Firmware restart, F–9
- Firmware restart address, F–93
- FIXUP console routine, 26–64 (III)
 - PALcode switching and, 27–7 (III)
 - procedure descriptor for, 26–70 (III)
 - using, 26–71 (III)
- FLOAT_REGISTER_MASK, F–24
- Floating-point branch instructions, 4–99 (I)
- Floating-point computational models, 4–68 (I)
- Floating-point control quadword, B–4
- Floating-point control register (FPCR)
 - accessing, 4–82 (I)
 - bit descriptions, 4–80 (I)
 - EXCB instruction with, A–15
 - instructions to read/write, 4–108 (I)
 - operate instructions that use, 4–101 (I)
 - processor initialization and, 4–83 (I)
 - saving and restoring, 4–83 (I)
 - trap disable bits in, 4–79 (I)
- Floating-point convert instructions, 3–13 (I)
 - Fa field requirements, 3–13 (I)
- Floating-point disabled fault, 14–10 (II-A)
 - service routine entry point, 14–25 (II-A)
- Floating-point division, performance impact of, A–12
- Floating-point enable (FEN) register
 - clearing, 10–10 (II-A)
 - defined, 15–3 (II-B), 20–3 (II-C)
 - described, 13–11 (II-A)
 - HWPCB and, 12–2 (II-A)
 - HWPCB, initial and, 27–25 (III)
 - PALcode switching and, 27–8 (III)
 - privileged context, 10–90 (II-A)
 - process context and, 18–1 (II-B), 23–1 (II-C)
 - processor initialization and, 27–23 (III)
- Floating-point format, number representation (encodings), 4–66 (I)
- Floating-point instructions
 - branch, 4–99 (I)
 - faults, 4–63 (I)
 - function field format, 4–84 (I)
 - introduced, 4–63 (I)
 - memory format, 4–90 (I)
 - opcodes and format summarized, C–1
 - operate, 4–101 (I)
 - rounding modes, 4–67 (I)
 - terminology, 4–64 (I)
 - trapping modes, 4–70 (I)
 - traps, 4–63 (I)
- Floating-point load instructions, 4–90 (I)
 - load F_floating, 4–91 (I)
 - load G_floating, 4–92 (I)
 - load S_floating, 4–93 (I)
 - load T_floating, 4–94 (I)
 - non-finite values and, 4–90 (I)

- Floating-point operate instructions, 4-101 (I)
 - add (IEEE), 4-110 (I)
 - add (VAX), 4-109 (I)
 - compare (IEEE), 4-112 (I)
 - compare (VAX), 4-111 (I)
 - conditional move, 4-106 (I)
 - convert IEEE floating to integer, 4-116 (I)
 - convert integer to IEEE floating, 4-117 (I)
 - convert integer to integer, 4-105 (I)
 - convert integer to VAX floating, 4-114 (I)
 - convert S_floating to T_floating, 4-118 (I)
 - convert T_floating to S_floating, 4-119 (I)
 - convert VAX floating to integer, 4-113 (I)
 - convert VAX floating to VAX floating, 4-115 (I)
 - copy sign, 4-104 (I)
 - divide (IEEE), 4-121 (I)
 - divide (VAX), 4-120 (I)
 - format of, 3-12 (I)
 - integer moves, from, 4-124 (I)
 - integer moves, to, 4-122 (I)
 - move from/to FPCR, 4-108 (I)
 - multiply (IEEE), 4-127 (I)
 - multiply (VAX), 4-126 (I)
 - subtract (IEEE), 4-131 (I)
 - subtract (VAX), 4-130 (I)
 - unused function codes with, 3-12 (I)
- Floating-point registers, 3-2 (I)
 - PALcode switching and, 27-8 (III)
 - See also Registers
- Floating-point single-precision operations, 4-63 (I)
- Floating-point store instructions, 4-90 (I)
 - non-finite values and, 4-90 (I)
 - store F_floating, 4-95 (I)
 - store G_floating, 4-96 (I)
 - store S_floating, 4-97 (I)
 - store T_floating, 4-98 (I)
- Floating-point support
 - floating-point control (FP_C) quadword, B-4
 - IEEE, 2-6 (I)
 - IEEE standard 754-1985, 4-89 (I)
 - instruction overview, 4-63 (I)
 - longword integer, 2-10 (I)
 - operate instructions, 4-101 (I)
 - optional, 4-2 (I)
 - quadword integer, 2-11 (I)
 - rounding modes, 4-67 (I)
 - single-precision operations, 4-63 (I)
 - trap modes, 4-70 (I)
 - VAX, 2-3 (I)
- Floating-point to integer move, 3-13 (I), 4-122 (I)

- Floating-point trapping modes, 4-70 (I)
 - See also Arithmetic traps
- FNOP code form, A-13
- FOE. See Fault on execute
- FOR. See Fault on read
- FOW. See Fault on write
- FP. See Frame pointer
- FP_C quadword, B-4
- FPCR. See Floating-point control register
- Frame pointer (FP) register, linkage for, 15-1 (II-B), 20-1 (II-C)
- FRU. See Field replaceable unit
- FTOIS instruction, 4-122 (I)
- FTOIT instruction, 4-122 (I)
- Function codes
 - IEEE floating-point, C-7
 - independent floating-point, C-9
 - numerical order listing, C-12
 - VAX floating-point, C-9
 - See also Opcodes

G

- G_floating data type, 2-4 (I)
 - alignment of, 2-5 (I)
 - mapping, 2-4 (I)
 - MAX/MIN, 4-66 (I)
 - unaligned data and, 14-26 (II-A)
- General class exceptions, F-23
 - common dispatch of, F-28
- General exception address (GENERAL_ENTRY) register, F-7
- GENTRAP (PALcode) instruction, 10-11 (II-A)
 - required recognition of, 6-4 (I)
 - trap information, 14-15 (II-A)
- gentrap (PALcode) instruction, 16-6 (II-B), 21-6 (II-C), F-86
 - raises software exceptions, F-26
 - required recognition of, 6-4 (I)
- GET_ENV variable routine, 26-60 (III)
- GETC terminal routine, 26-34 (III)
 - ISO Latin-1 support and, 25-4 (III)
- GH. See Granularity hint
- Global pointer (GP) register, linkage for, 15-1 (II-B), 20-1 (II-C)
- Global translation hint, F-16

Granularity hint (GH)
 bits in PTE, F-16
 block in HWRPB, 26-13 (III)
 fields in, 26-14 (III)
GRAPHICS, system variation field, 26-12 (III)

H

HAL (Hardware abstraction layer), F-3
HALT (PALcode) instruction
 required, 6-6 (I)
 state transitions and, 27-1 (III)
halt (PALcode) instruction, F-47
 required, 6-6 (I)
 writes PAL_BASE register, F-8
 See also reboot (PALcode) instruction
Halt PCBB register, per-CPU slot field for, 26-18 (III)
Halt processor, per-CPU slot fields for, 26-18 (III)
Halt requested, per-CPU state flag, 26-22 (III)
 multiprocessor booting and, 27-27 (III)
Hardware abstraction layer
 interfaces for, F-3
Hardware context, 18-1 (II-B), 23-1 (II-C)
Hardware errors, when unrecoverable, F-28
Hardware interrupts, F-31
 interprocessor, 14-19 (II-A)
 interval clock, 14-19 (II-A)
 powerfail, 14-20 (II-A)
 servicing, 19-7 (II-B), 24-7 (II-C)
Hardware nonprivileged context, 12-2 (II-A)
Hardware privileged context, 12-2 (II-A)
 switching, 12-2 (II-A)
Hardware privileged context block (HPCB)
 process unique value in, 10-78 (II-A)
 swapping ownership of, 10-90 (II-A)
Hardware privileged context block (HWPCB)
 cold booting and, 27-25 (III)
 format, 12-2 (II-A)
 original built by HWRPB, 12-4 (II-A)
 PCBB register, 13-16 (II-A)
 specified by PCBB, 12-2 (II-A)
 warm booting and, 27-26 (III)
 writing to, 12-3 (II-A)
Hardware restart parameter block (HWRPB), 26-1 (III)
 cold boot and, 27-9 (III)
 discontiguous data structures, 26-2 (III)

 fields for, 26-6 (III)
 interval clock interrupt, 14-19 (II-A)
 loading at cold boot, 27-17 (III)
 logout area, 14-23 (II-A)
 overview of, 26-2 (III)
 size field in, 26-6 (III)
 structure of, 26-4 (III)

HIGH_LEVEL, IRQ table index name, F-5
HWPCB. See Hardware privileged context block
HWRPB. See Hardware restart parameter block

I

I/O access, nonmapped, F-13
I/O device interrupts, 14-19 (II-A)
I/O device registers, at power-up initialization, 27-4 (III)
I/O devices
 device-specific operations for, 26-51 (III)
 generic, closing for access, 26-50 (III)
 generic, opening for access, 26-48 (III)
 generic, reading from, 26-53 (III)
 generic, routines for, 26-46 (III)
 generic, writing to, 26-55 (III)
 required implementation support for, 26-48 (III)
 service routine entry points, 14-28 (II-A)
I/O devices, DMA
 MB and WMB with, 5-23 (I)
 reliably communicating with processor, 5-27 (I)
 shared memory locations with, 5-11 (I)
I/O interface overview, 8-1 (I)
I/O support, HAL interface for, F-3
IEEE floating-point
 exception handlers, B-2
 floating-point control (FP_C) quadword, B-4
 format, 2-6 (I)
 FPCR (floating-point control register), 4-80 (I)
 function field format, 4-85 (I)
 hardware support, B-1
 high-performance arithmetic, 4-69 (I)
 inexact exceptions, 4-69 (I)
 NaN, 2-6 (I)
 options, B-1
 precise exceptions, 4-72 (I)
 S_floating, 2-6 (I)
 standard charts, B-11
 standard, mapping to, B-6
 standards compliance, 4-69 (I)

- T_floating, 2–8 (I)
- trap handling, B–6
- X_floating, 2–9 (I)
- See also Floating-point instructions
- IEEE floating-point control quadword, B–4
- IEEE floating-point instructions
 - add, 4–110 (I)
 - compare, 4–112 (I)
 - convert from integer, 4–117 (I)
 - convert S_floating to T_floating, 4–118 (I)
 - convert T_floating to S_floating, 4–119 (I)
 - convert to integer, 4–116 (I)
 - divide, 4–121 (I)
 - function codes for, C–7
 - integer moves, from, 4–124 (I)
 - multiply, 4–127 (I)
 - operate, 4–101 (I)
 - register moves, to, 4–122 (I)
 - square root, 4–129 (I)
 - subtract, 4–131 (I)
- IEEE standard, 4–89 (I)
 - conformance to, B–1
 - mapping to, B–6
- IEEE trapping modes, 4–72 (I)
 - /SU, 4–72 (I)
 - /SUI, 4–73 (I)
 - /SV, 4–72 (I)
 - /SVI, 4–73 (I)
 - /U, 4–72 (I)
 - /V, 4–72 (I)
 - default mode, 4–72 (I)
 - precise, 4–72 (I)
 - summary, 4–73 (I)
- IEEE, subsetted instruction exception, F–27
- IEEE-compliant arithmetic, 4–69 (I)
- IGN (ignore), 1–9 (I)
- IKSP register. See Kernel stack pointer, initial
- Illegal instruction exceptions, F–25
- Illegal instruction trap, 14–15 (II-A)
 - service routine entry point, 14–26 (II-A)
- Illegal operand trap, service routine entry point, 14–26 (II-A)
- Illegal PALcode operand trap, 14–15 (II-A)
- IMB (PALcode) instruction, 5–24 (I)
 - required, 6–7 (I)
 - virtual I-cache coherency, 5–5 (I)
- imb (PALcode) instruction, F–87
 - required, 6–7 (I)
- IMB, HWPCB bit, 12–3 (II-A)
- IMB, PCB bit, 18–3 (II-B), 23–3 (II-C)
- IMP (implementation dependent), 1–9 (I)
- IMP_DATA_PA
 - memory data descriptor table field, 27–12 (III)
 - null memory cluster descriptor field, 27–13 (III)
- IMPLVER (Implementation version) instruction, 4–141 (I)
 - value assignments, D–5
- Independent floating-point function codes, C–9
- INE bit
 - exception summary parameter, 14–12 (II-A)
 - exception summary register, 19–5 (II-B), 24–5 (II-C), F–25
 - See also Arithmetic traps, inexact result
- INED bit. See Trap disable bits, inexact result trap
- Inexact result bit, exception summary register, F–25
- Inexact result enable (INEE)
 - FP_C quadword bit, B–5
- Inexact result status (INES)
 - FP_C quadword bit, B–5
- Inexact result trap, 14–13 (II-A), 19–5 (II-B), 24–5 (II-C), F–25
- Infinity, 4–65 (I)
 - conversion to integer, 4–89 (I)
- Initialization, PALcode environment, F–91
- initpal (PALcode) instruction, F–48, F–50
 - initialization and, F–92
 - reads PAL_BASE register, F–8
 - writes KGP register, F–7
 - writes PCR register, F–8
 - writes PDR register, F–8
- initpcr (PALcode) instruction, F–50
- INSBL instruction, 4–56 (I)
- Insert byte instructions, 4–56 (I)
 - big-endian support with, 2–13 (I)
- Insert into queue PALcode instructions
 - longword, 10–45 (II-A)
 - longword at head interlocked, 10–29 (II-A)
 - longword at head interlocked resident, 10–31 (II-A)
 - longword at tail interlocked, 10–37 (II-A)
 - longword at tail interlocked resident, 10–39 (II-A)
 - quadword, 10–47 (II-A)
 - quadword at head interlocked, 10–33 (II-A)
 - quadword at head interlocked resident, 10–35

- (II-A)
 - quadword at tail interlocked, 10–41 (II-A)
 - quadword at tail interlocked resident, 10–43 (II-A)
- INSLH instruction, 4–56 (I)
- INSLL instruction, 4–56 (I)
- INSQH instruction, 4–56 (I)
- INSQHIL (PALcode) instruction, 10–29 (II-A)
- INSQHILR (PALcode) instruction, 10–31 (II-A)
- INSQHIQ (PALcode) instruction, 10–33 (II-A)
- INSQHIQR (PALcode) instruction, 10–35 (II-A)
- INSQL instruction, 4–56 (I)
- INSQTIL (PALcode) instruction, 10–37 (II-A)
- INSQTILR (PALcode) instruction, 10–39 (II-A)
- INSQTIQ (PALcode) instruction, 10–41 (II-A)
- INSQTIQR (PALcode) instruction, 10–43 (II-A)
- INSQUEL (PALcode) instruction, 10–45 (II-A)
- INSQUEL/D (PALcode) instruction, 10–45 (II-A)
- INSQUEQ (PALcode) instruction, 10–47 (II-A)
- INSQUEQ/D (PALcode) instruction, 10–47 (II-A)
- Instances of system software, 27–12 (III)
- Instruction encodings
 - common architecture, C–2
 - numerical order, C–12
 - opcodes and format summarized, C–1
- Instruction fault entry (entIF) register, 15–2 (II-B), 19–4 (II-B), 19–6 (II-B), 20–2 (II-C), 24–4 (II-C), 24–7 (II-C)
- Instruction fault, system entry for, 19–4 (II-B), 24–4 (II-C)
- Instruction fetches (memory), 5–12 (I)
- Instruction formats
 - branch, 3–11 (I)
 - conventions, 3–9 (I)
 - floating-point convert, 3–13 (I)
 - floating-point operate, 3–12 (I)
 - floating-point to integer move, 3–13 (I)
 - illegal trap, 14–15 (II-A)
 - memory, 3–10 (I)
 - memory jump, 3–10 (I)
 - operand values, 3–9 (I)
 - operators, 3–6 (I)
 - overview, 1–3 (I)
 - PALcode, 3–13 (I)
 - registers, 3–1 (I)

- Instruction set
 - access type field, 3–5 (I)
 - Boolean, 4–42 (I)
 - branch, 4–19 (I)
 - byte manipulate, 4–48 (I)
 - conditional move (integer), 4–44 (I)
 - data type field, 3–6 (I)
 - floating-point subsetting, 4–2 (I)
 - integer arithmetic, 4–25 (I)
 - introduced, 1–6 (I)
 - jump, 4–19 (I)
 - load memory integer, 4–4 (I)
 - miscellaneous, 4–132 (I)
 - multimedia, 4–154 (I)
 - name field, 3–5 (I)
 - opcode qualifiers, 4–3 (I)
 - operand notation, 3–4 (I)
 - overview, 4–1 (I)
 - shift, arithmetic, 4–47 (I)
 - software emulation rules, 4–2 (I)
 - store memory integer, 4–4 (I)
 - VAX compatibility, 4–152 (I)
 - See also Floating-point instructions
- Instruction stream translation buffer (ITB), 26–13 (III)
- Instruction stream. See I-stream
- Instructions, overview, 1–4 (I)
- INSWH instruction, 4–56 (I)
- INSWL instruction, 4–56 (I)
- Integer division, A–12
- Integer overflow bit, exception summary register, F–25
- Integer overflow trap, 14–14 (II-A), 19–5 (II-B), 24–5 (II-C), F–25
- Integer registers
 - defined, 3–1 (I)
 - PALcode switching and, 27–8 (III)
 - R31 restrictions, 3–1 (I)
 - See also Registers
- INTEGER_REGISTER_MASK, F–24
- Internal processor registers (IPR)
 - address space number, 13–4 (II-A), F–7
 - AST enable, 13–5 (II-A)
 - AST summary, 13–7 (II-A)
 - CALL_PAL MFPR with, 13–1 (II-A)
 - CALL_PAL MTPR with, 13–1 (II-A)
 - data alignment trap fixup, 13–9 (II-A)
 - defined, 9–1 (II-A)
 - executive stack pointer, 13–10 (II-A)

- floating-point enable, 13–11 (II-A)
- general exception address, F–7
- interprocessor interrupt request, 13–12 (II-A)
- interrupt exception address, F–7
- interrupt priority level, 13–13 (II-A)
- kernel global pointer, F–7
- kernel mode with, 13–1 (II-A)
- kernel stack pointer (IKSP), initial, F–7
- machine check error summary, 13–14 (II-A)
- machine check error summary (MCES) register, F–7
- memory management exception, F–8
- MFPR instruction with, 10–85 (II-A)
- MTPR instruction with, 10–86 (II-A)
- page directory base, F–8
- page table base, 13–18 (II-A)
- PALcode image base address, F–8
- panic exception, F–8
- performance monitoring, 13–15 (II-A)
- privileged context block base, 13–16 (II-A)
- process control region base, F–8
- processor base, 13–17 (II-A)
- processor status, F–8
- restart execution address, F–8
- returning state of, F–57
- software interrupt request, 13–20 (II-A), F–9
- software interrupt summary, 13–21 (II-A)
- summarized, 13–2 (II-A), F–6
- supervisor stack pointer, 13–22 (II-A)
- system control block base, 13–19 (II-A)
- system page table base, 13–23 (II-A)
- system service exception address, F–9
- thread environment block base, F–9
- thread unique value, F–9
- translation buffer check, 13–24 (II-A)
- translation buffer invalidate all, 13–25 (II-A)
- translation buffer invalidate all process, 13–26 (II-A)
- translation buffer invalidate single, 13–27 (II-A)
- user stack pointer, 13–28 (II-A)
- virtual address boundary, 13–29 (II-A)
- virtual page base, 13–30 (II-A)
- Who-Am-I, 13–31 (II-A)
- Interprocessor console communications, 26–75 (III)
- Interprocessor interrupt, 14–19 (II-A)
 - generating, 16–29 (II-B), 21–28 (II-C)
 - protocol for, 14–19 (II-A)
 - service routine entry point, 14–27 (II-A)
- Interprocessor interrupt request (IPIR) register
 - described, 13–12 (II-A)
 - protocol for, 14–19 (II-A)
- Interrupt acknowledge, F–33
- Interrupt dispatch
 - example, F–31
 - table (IDT), F–31
- Interrupt enable mask, F–30
- Interrupt entry (entInt) register, 15–2 (II-B), 19–4 (II-B), 19–7 (II-B), 20–2 (II-C), 24–4 (II-C), 24–7 (II-C)
- Interrupt exception address (INTERRUPT_ENTRY) register, F–7
- Interrupt handling
 - HAL interface for, F–3
- Interrupt level table (ILT), F–30
 - index values/names for, F–5
- Interrupt mask table (IMT), F–30
- Interrupt pending (IP) field, in PS register, 14–6 (II-A)
- Interrupt priority level (IPL), 14–6 (II-A)
 - events associated with, 14–17 (II-A)
 - field in PS register, 14–5 (II-A)
 - hardware levels, 14–6 (II-A)
 - kernel mode software with, 14–17 (II-A)
 - operation of, 14–16 (II-A)
 - PALcode switching and, 27–8 (III)
 - processor initialization and, 27–24 (III)
 - PS with, 19–2 (II-B), 24–2 (II-C)
 - recording pending software (SISR register), 13–21 (II-A)
 - requesting software (SIRR register), 13–20 (II-A)
 - service routine entry points, 14–27 (II-A)
 - software interrupts, 14–18 (II-A)
 - software levels, 14–6 (II-A)
 - See also Interrupt priority level (IPL) register
- Interrupt priority level (IPL) register
 - described, 13–13 (II-A)
 - interrupt arbitration, 14–32 (II-A)
 - See also Interrupt priority level (IPL)
- Interrupt request levels (IRQI)
 - ILT table for, F–30
 - PSR and, F–5
 - PSR and di instruction, F–42
 - swapping, F–68
- Interrupt service routines
 - entry point, 14–24 (II-A)
 - in each process, 14–17 (II-A)
 - introduced, 14–16 (II-A)
- Interrupt tables (IDT, ILT, IMT), F–9

Interrupt tables at initialization, F-92

Interrupt trap frame, building, F-32

Interrupt vectors, mask table for, F-30

Interrupts, F-30

- actions, summarize, 14-2 (II-A)
- disabling, F-42
- enabling, F-46
- hardware arbitration, 14-32 (II-A)
- I/O device, 14-19 (II-A)
- initiated by PALcode, 14-29 (II-A)
- initiation, 14-17 (II-A)
- instruction completion, 14-16 (II-A)
- interprocessor, 14-19 (II-A)
- introduced, 14-16 (II-A)
- PALcode arbitration, 14-32 (II-A)
- passive release, 14-19 (II-A)
- powerfail, 14-20 (II-A)
- processor state transitions, 14-34 (II-A)
- processor status register and, F-5
- program counter value, 14-2 (II-A)
- returning from, F-63
- software, 14-17 (II-A)
- software requests for, F-33
- sources for, 19-2 (II-B), 24-2 (II-C)
- stack frames for, 14-7 (II-A), 19-3 (II-B), 24-3 (II-C)
- system entry for, 19-4 (II-B), 24-4 (II-C)

Interval clock interrupt, 14-19 (II-A)

- HWRPB field for, 26-7 (III)
- service routine entry point, 14-27 (II-A)

intr_flag register, 15-3 (II-B), 20-3 (II-C)

- cleared by retsys, F-62
- cleared by rfe, F-64

INV bit

- exception summary parameter, 14-12 (II-A)
- exception summary register, 19-6 (II-B), 24-6 (II-C), F-25
- See also Arithmetic traps, invalid operation

Invalid address exceptions, F-26

Invalid operation bit, exception summary register, F-25

Invalid operation enable (INVE)

- FP_C quadword bit, B-6

Invalid operation status (INVS)

- FP_C quadword bit, B-5

Invalid operation trap, 14-13 (II-A), 19-6 (II-B), 24-6 (II-C), F-25

INVD bit. See Trap disable bits, invalid operation

IOCTL console device routine, 26-51 (III)

IOV bit

- exception summary parameter, 14-12 (II-A)
- exception summary register, 19-5 (II-B), 24-5 (II-C), F-25
- See also Arithmetic traps, integer overflow

IPI_LEVEL, IRQL table index name, F-5

IPL. See Interrupt priority level

IPR. See Internal processor registers (IPR)

IPR_KSP (internal processor register kernel stack pointer), 13-1 (II-A)

IRQL

- See Interrupt request levels
- See also rdirql and swpirql

ISO Latin-1 support, 25-4 (III)

- PROCESS_KEYCODE and, 26-42 (III)

I-stream

- coherency of, 6-7 (I)
- design considerations, A-2
- modifying physical, 5-5 (I)
- modifying virtual, 5-5 (I)
- PALcode with, 6-2 (I)
- with caches, 5-5 (I)

ITB. See Instruction stream translation buffer

ITOFF instruction, 4-124 (I)

ITOFS instruction, 4-124 (I)

ITOFT instruction, 4-124 (I)

J

JMP instruction, 4-23 (I)

JSR instruction, 4-23 (I)

JSR_COROUTINE instruction, 4-23 (I)

Jump instructions, 4-19 (I), 4-23 (I)

- branch prediction logic, 4-23 (I)
- coroutine linkage, 4-24 (I)
- lock_flag with, 4-10 (I)
- return from subroutine, 4-23 (I)
- unconditional long jump, 4-24 (I)
- See also Control instructions

K

kbpt (PALcode) instruction, F-88

Kernel global pointer (KGP) register, 15-3 (II-B), 20-3 (II-C), F-7

- initialization, F-92
- initializing, F-48

Kernel read enable (KRE)

- access control violation (ACV) fault and, 11–16 (II-A)
- bit in PTE, 11–4 (II-A), 17–4 (II-B), 22–4 (II-C)
- Kernel stack, F–10
 - under/overflow detection, F–89
- Kernel stack pointer (IKSP), initial, F–7
 - context switching and, F–11, F–67
 - initializing, F–48
 - returning contents of, F–53
 - swapping to current, F–69
 - trap frames and, F–21
- Kernel stack pointer (KSP) register
 - defined, 15–3 (II-B), 20–3 (II-C)
 - HWPCB and, 12–2 (II-A)
 - HWPCB, initial and, 27–25 (III)
 - PALcode switching and, 27–8 (III)
 - process context and, 18–1 (II-B), 23–1 (II-C)
 - processor initialization and, 27–24 (III)
- Kernel stack, PALcode access to, 14–28 (II-A)
- Kernel stack, when corrupted, F–29
- Kernel write enable (KWE)
 - bit in PTE, 11–4 (II-A), 17–4 (II-B), 22–4 (II-C)
- KERNEL_BREAKPOINT breakpoint type, F–27
- Keycode, translating, 26–42 (III)
- KGP. See Kernel global pointer
- Kseg
 - format of, 17–2 (II-B), 22–2 (II-C)
 - mapping of, 17–1 (II-B), 22–1 (II-C)
 - physical space with, 17–3 (II-B), 22–3 (II-C)
 - virtual address format, 17–3 (II-B)
- KSP. See Kernel stack pointer

L

- LANGUAGE environment variable, 26–28 (III)
- Languages, supported by console, 26–28 (III)
- LDA instruction, 4–5 (I)
- LDAH instruction, 4–5 (I)
- LDBU instruction, 4–6 (I)
 - big-endian support with, 2–13 (I)
- LDF instruction, 4–91 (I)
 - big-endian support with, 2–13 (I)
 - unaligned data and, 14–26 (II-A)
- LDG instruction, 4–92 (I)
 - unaligned data and, 14–26 (II-A)
- LDL instruction, 4–6 (I)
 - big-endian support with, 2–13 (I)
 - unaligned data and, 14–26 (II-A)
- LDL_L instruction, 4–9 (I)
 - big-endian support with, 2–13 (I)
 - processor lock register/flag and, 4–10 (I)
 - restrictions, 4–10 (I)
 - STx_C instruction and, 4–9 (I)
- LDQ instruction, 4–6 (I)
 - unaligned data and, 14–26 (II-A)
- LDQ_L instruction, 4–9 (I)
 - processor lock register/flag and, 4–10 (I)
 - restrictions, 4–10 (I)
 - STx_C instruction and, 4–10 (I)
 - unaligned data and, 14–26 (II-A)
- LDQ_U instruction, 4–8 (I)
- LDQP (PALcode) instruction, 10–84 (II-A)
- LDS instruction, 4–93 (I)
 - big-endian support with, 2–13 (I)
 - FPCR and, 4–84 (I)
 - unaligned data and, 14–26 (II-A)
- LDT instruction, 4–94 (I)
 - unaligned data and, 14–26 (II-A)
- LDWU instruction, 4–6 (I)
 - big-endian support with, 2–13 (I)
- LEFT_SHIFT(x,y) operator, 3–7 (I)
- lg operator, 3–7 (I)
- LICENSE environment variable, 26–27 (III)
- Literals, operand notation, 3–5 (I)
- Litmus tests, shared data veracity, 5–17 (I)
- Load instructions
 - emulation of, 4–2 (I)
 - FETCH instruction, 4–139 (I)
 - Load address, 4–5 (I)
 - Load address high, 4–5 (I)
 - load byte, 4–6 (I)
 - load longword, 4–6 (I)
 - load quadword, 4–6 (I)
 - load quadword locked, 4–10 (I)
 - load sign-extended longword locked, 4–9 (I)
 - load unaligned quadword, 4–8 (I)
 - load word, 4–6 (I)
 - multiprocessor environment, 5–6 (I)
 - serialization, 4–142 (I)
 - unaligned data and, 14–26 (II-A)
 - See also Floating-point load instructions
- Load literal, A–14

Load memory integer instructions, 4–4 (I)

LOAD_LOCKED operator, 3–7 (I)

Load-locked, defined, 5–16 (I)

Location, 5–11 (I)

Location access constraints, 5–14 (I)

Lock flag, per-processor
 defined, 3–2 (I)
 load locked instructions and, 4–10 (I)
 when cleared, 4–10 (I)

Lock registers, per-processor
 defined, 3–2 (I)
 load locked instructions and, 4–10 (I)
 STx_C and, 4–10 (I)

Lock variables, with WMB instruction, 4–151 (I)

lock_flag register, 4–10 (I), 15–3 (II-B), 20–3 (II-C)
 cleared by retsys, F–62
 cleared by rfe, F–64

Logical instructions. See Boolean instructions

Logout area, 14–23 (II-A)
 length, per-CPU slot field for, 26–18 (III)
 physical address, per-CPU slot field for, 26–18 (III)

Longword data type, 2–2 (I)
 alignment of, 2–11 (I)
 atomic access of, 5–2 (I)

LSB (least significant bit), defined for floating-point, 4–65 (I)

M

/M qualifier, IEEE minus infinity, 4–68 (I)

Machine check error handling, F–35

Machine check error summary (MCES) register
 defined, 15–3 (II-B), 20–3 (II-C)
 described, 13–14 (II-A), F–7
 format of, F–35
 PALcode switching and, 27–8 (III)
 processor initialization and, 27–24 (III)
 reading, 16–13 (II-B), 21–12 (II-C)
 returning contents of, F–54
 structure of, 19–7 (II-B), 24–7 (II-C)
 using, 14–22 (II-A)
 writing, 16–31 (II-B), 21–30 (II-C), F–79

Machine checks, 14–21 (II-A)
 actions, summarized, 14–2 (II-A)
 catastrophic conditions with, F–37
 classes of, F–34
 disabling during debug, F–36
 initiated by PALcode, 14–29 (II-A)
 interrupt entry for, 19–7 (II-B), 24–7 (II-C)
 logout area, 14–23 (II-A)
 masking, 14–21 (II-A)
 no disabling of, 14–21 (II-A)
 one per error, 14–22 (II-A)
 processor correctable, 14–21 (II-A)
 program counter (PC) value, 14–21 (II-A)
 REI instruction with, 14–22 (II-A)
 retry flag, 14–22 (II-A)
 service routine entry points, 14–27 (II-A)
 sources for, F–34
 stack frames for, 14–7 (II-A)
 system correctable, 14–21 (II-A)
 type codes, F–36
 unrecoverable reported, F–36

Machine checks service routines
 entry point, 14–24 (II-A)

Magtape bootstrap image
 ANSI format, 27–42 (III)
 boot blocked, 27–43 (III)

Major modes, 27–3 (III)

Major state transitions, 27–2 (III)
 console rules for, 27–2 (III)

Major states, 27–1 (III)

MAP_F function, 2–3 (I)

MAP_S function, 2–7 (I)

MAP_x operator, 3–7 (I)

Mask byte instructions, 4–58 (I)
 big-endian support with, 2–13 (I)

Masking, machine checks with, 14–21 (II-A)

MAX, defined for floating-point, 4–66 (I)

maxCPU, 15–2 (II-B), 20–2 (II-C)

Maximum ASN value, HWRPB field for, 26–6 (III)

MAXS(x,y) operator, 3–7 (I)

MAXSB8 instruction, 4–155 (I)

MAXSW4 instruction, 4–155 (I)

MAXU(x,y) operator, 3–8 (I)

MAXUB8 instruction, 4–155 (I)

MAXUW4 instruction, 4–155 (I)

MB (Memory barrier) instruction, 4–142 (I)
 DMA I/O and, 5–23 (I)
 LDx_L/STx_C and, 4–15 (I)
 multiprocessor D-stream and, 5–22 (I)
 multiprocessors only, 4–142 (I)
 shared data structures and, 5–9 (I)

- WMB compared to, 4–151 (I)
- See also IMB, WMB
- MBZ (must be zero), 1–8 (I)
- MCES. See Machine check error summary
- MCK bit, machine check error summary register, 13–14 (II-A), F–36
- MEMC. See Memory cluster descriptor
- MEMDSC. See Memory data descriptor table
- Memory access
 - aligned byte/word, A–11
 - coherency of, 5–1 (I)
 - granularity of, 5–2 (I)
 - width of, 5–3 (I)
 - WMB instruction and, 4–150 (I)
- Memory alignment, requirement for, 5–2 (I)
- Memory barrier instructions. See MB, IMB (PALcode), and WMB instructions
- Memory barriers, 5–22 (I)
- Memory cluster descriptor, 27–9 (III)
 - distributed format, 27–15 (III)
 - distributed linking, 27–17 (III)
 - passing to system software, 27–10 (III)
 - static, 27–10 (III)
 - static fields, 27–12 (III)
 - static format, 27–11 (III)
- Memory clusters, 27–9 (III)
 - distributed, 27–12 (III)
 - static, 27–10 (III)
- Memory data descriptor (MEMDSC) table
 - distributed memory clusters and, 27–10 (III)
 - fields, 27–11 (III)
 - format, 27–11 (III)
 - null memory cluster field, 27–13 (III)
 - offset, HWRPB field for, 26–8 (III)
 - static memory clusters and, 27–10 (III)
 - warm booting and, 27–26 (III)
- Memory format instructions
 - opcodes and format summarized, C–1
- Memory instruction format, 3–10 (I)
- Memory jump instruction format, 3–10 (I)
- Memory management
 - address translation, 11–8 (II-A)
 - control of, 11–3 (II-A), 17–3 (II-B), 22–3 (II-C)
 - faults, 11–15 (II-A), 14–9 (II-A), 17–13 (II-B), 22–13 (II-C)
 - interrupts and, 14–17 (II-A)
 - introduced, 11–1 (II-A)
 - multiprocessors and, 11–6 (II-A)
 - page frame number (PFN), 11–6 (II-A)
 - page table entry (PTE), 11–3 (II-A)
 - process context and, 12–1 (II-A)
 - protection, 11–7 (II-A)
 - protection code, 11–7 (II-A)
 - PTE modified by software, 11–6 (II-A)
 - support in PALcode, 6–2 (I)
 - translation buffers and, 11–13 (II-A)
 - unrecoverable error, 14–21 (II-A)
 - See also Address translation
- Memory management exception
 - (MEM_MGMT_ENTRY) register, F–8
- Memory management fault entry (entMM) register, 15–2 (II-B), 19–4 (II-B), 19–8 (II-B), 20–2 (II-C), 24–4 (II-C), 24–8 (II-C)
- Memory management faults
 - registers used, 14–9 (II-A)
 - system entry for, 19–4 (II-B), 24–4 (II-C)
 - types, 17–13 (II-B), 22–13 (II-C)
 - unaligned data, 14–14 (II-A)
- Memory prefetch registers
 - defined, 3–3 (I)
- Memory protection, 17–6 (II-B), 22–6 (II-C)
- Memory sizing at cold boot, 27–9 (III)
- Memory testing at cold boot, 27–9 (III)
- Memory-like behavior, 5–3 (I)
- MF_FPCR instruction, 4–108 (I)
- MFPR_IPR_name (PALcode) instruction, 10–85 (II-A)
- MIN, defined for floating-point, 4–66 (I)
- MINS(x,y) operator, 3–8 (I)
- MINSB8 instruction, 4–155 (I)
- MINSW4 instruction, 4–155 (I)
- MINU(x,y) operator, 3–8 (I)
- MINUB8 instruction, 4–155 (I)
- MINUW4 instruction, 4–155 (I)
- MIP bit, machine check error summary register, 19–8 (II-B), 24–8 (II-C)
- Miscellaneous instructions, 4–132 (I)
- MMCSR code, 17–13 (II-B), 22–13 (II-C)
- Modify intent, prefetch with, A–10
- MOP-based network bootstrapping, 27–45 (III)
- Move instructions (conditional). See Conditional move instructions

Move, register-to-register, A-15

MPCAP, system variation field, 26-13 (III)

MSKBL instruction, 4-58 (I)

MSKLN instruction, 4-58 (I)

MSKLL instruction, 4-58 (I)

MSKQL instruction, 4-58 (I)

MSKWH instruction, 4-58 (I)

MSKWL instruction, 4-58 (I)

MT_FPCR instruction, 4-108 (I)
synchronization requirement, 4-83 (I)

MTPR_IPR_name (PALcode) instruction, 10-86 (II-A)

MULF instruction, 4-126 (I)

MULG instruction, 4-126 (I)

MULL instruction, 4-35 (I)
MULQ and, 4-35 (I)

MULQ instruction, 4-36 (I)
MULL and, 4-35 (I)
UMULH and, 4-36 (I)

MULS instruction, 4-127 (I)

MULT instruction, 4-127 (I)

Multimedia instructions, 4-154 (I)

Multiple instruction issue, A-3

Multiply instructions
multiply longword, 4-35 (I)
multiply quadword, 4-36 (I)
multiply unsigned quadword high, 4-37 (I)
See also Floating-point operate

Multiprocessor bootstrapping, 27-27 (III)

Multiprocessor environment
booting, 27-27 (III)
cache coherency in, 5-6 (I)
console requirements, 26-25 (III)
context switching, 5-24 (I)
interprocessor interrupt, 14-19 (II-A)
I-stream reliability, 5-24 (I)
MB and WMB with, 5-23 (I)
memory faults, 14-10 (II-A)
memory management in, 11-6 (II-A)
move operations in, 10-73 (II-A)
no implied barriers, 5-22 (I)
read/write ordering, 5-10 (I)
serialization requirements in, 4-142 (I)
shared data, 5-6 (I), A-6

Multithread implementation, 10-78 (II-A)

N

NaN (Not-a-Number)
conversion to integer, 4-89 (I)
copying, generating, propagating, 4-89 (I)
copying, generating, propagating, 4-89 (I)
defined, 2-6 (I)
quiet, 4-65 (I)
signaling, 4-65 (I)

NATURALLY ALIGNED data objects, 1-8 (I)

Negate stylized code form, A-15

Network bootstrapping, 27-45 (III)

New PALcode, 27-5 (III)

Next PC, 14-2 (II-A)

Non-finite number, 4-65 (I)

Nonmapped address space, F-13

Nonmemory-like behavior, 5-3 (I)

NOP, universal (UNOP), A-13

Normal prefetch, A-10

NOT instruction, 4-43 (I)

NOT operator, 3-8 (I)

NOT stylized code form, A-15

O

OFFSET
distributed memory cluster descriptor field,
27-15 (III)

Opcode qualifiers
default values, 4-3 (I)
notation, 4-3 (I)
See also specific qualifiers

Opcodes
Alpha Linux PALcode, C-20
common architecture, C-2
numerical order listing, C-12
OpenVMS PALcode, C-16
reserved, C-25
summary, C-10
Tru64 UNIX PALcode, C-19
unused function codes for, C-25
See also Function codes

opDec, 15-2 (II-B), 20-2 (II-C)

OPEN device routine, 26-48 (III)
determines WRITE characteristics, 26-56 (III)

OpenVMS PALcode instructions (list), 10-1 (II-A)

OpenVMS PALcode, instruction summary, C-16

- Operand expressions, 3–4 (I)
- Operand notation, 3–4 (I)
- Operand values, 3–4 (I)
- Operate instruction format
 - unused function codes with, 3–11 (I)
- Operate instructions
 - convert with integer overflow, 4–79 (I)
 - opcodes and format summarized, C–1
- Operator halted (OH) flag, 27–39 (III)
 - multiprocessor booting and, 27–27 (III)
 - per-CPU state contains, 26–22 (III)
- Operators, instruction format, 3–6 (I)
- Optimization. See Performance optimizations
- OR operator, 3–8 (I)
- ORNOT instruction, 4–43 (I)
- OS Loader, F–3
- Overflow bit, exception summary register, F–25
- Overflow enable (OVFE)
 - FP_C quadword bit, B–6
- Overflow status (OVFS)
 - FP_C quadword bit, B–5
- Overflow trap, 14–13 (II-A), 19–5 (II-B), 24–5 (II-C), F–25
- Overlap
 - location access constraints, 5–14 (I)
 - processor issue constraints, 5–13 (I)
 - visibility, 5–14 (I)
- OVF bit
 - exception summary parameter, 14–12 (II-A)
 - exception summary register, 19–5 (II-B), 24–5 (II-C), F–25
 - See also Arithmetic traps, overflow
- OVFD bit. See Trap disable bits, overflow disable

P

- Pack to bytes instructions, 4–158 (I)
- Page directory base (PDR) register, F–8
 - context switching and, F–71
 - initializing, F–48
 - maps PTEs, F–15
- Page directory entry (PDE), F–15
- Page frame number (PFN)
 - address translation and, 11–8 (II-A)
 - bits in PTE, 11–4 (II-A), 17–4 (II-B), 22–4 (II-C), F–14, F–16

- context switching and, F–12, F–67
 - determining validation, 11–6 (II-A)
 - finding for SCB, 13–19 (II-A)
 - hardware context switching and, 12–2 (II-A)
 - physical address translation and, 17–7 (II-B), 22–7 (II-C)
 - PTBR register, 13–18 (II-A)
 - when a PDR, F–15
- Page size, HWRPB field for, 26–6 (III)
- Page sizes, 17–2 (II-B), 22–2 (II-C)
- Page table base (PTBR) register, 13–18 (II-A)
 - address translation and, 11–8 (II-A)
 - defined, 15–4 (II-B), 20–4 (II-C)
 - HWPCB and, 12–2 (II-A)
 - HWPCB, initial and, 27–25 (III)
 - PALcode switching and, 27–8 (III)
 - physical address translation and, 17–7 (II-B), 22–7 (II-C)
 - privileged context, 10–90 (II-A)
 - process context and, 18–1 (II-B), 23–1 (II-C)
 - processor initialization and, 27–24 (III)
 - SYSPTBR and, 11–11 (II-A)
- Page table entry (PTE)
 - after software changes, 11–13 (II-A), 17–6 (II-B), 22–6 (II-C)
 - atomic modification of, 11–6 (II-A)
 - calculating at cold boot, 27–20 (III)
 - changing and managing, 17–6 (II-B), 22–6 (II-C)
 - format of, 11–3 (II-A), 17–3 (II-B), 22–3 (II-C), F–16
 - modified by software, 11–6 (II-A)
 - multiprocessors and, 11–6 (II-A)
 - page frame number (PFN) with, F–14
 - page protection, 11–7 (II-A)
 - virtual access of, 17–9 (II-B), 22–9 (II-C)
- Page table space
 - loading at cold boot, 27–18 (III)
- Page tables
 - calculating base, 27–20 (III)
 - initial mapping at cold boot, 27–20 (III)
 - physical traversal algorithm, F–15
 - traversing, F–14
- PAGES
 - distributed memory cluster descriptor field, 27–15 (III)
 - null memory cluster descriptor field, 27–14 (III)
 - static memory cluster descriptor field, 27–12 (III)
- Pages
 - collecting statistics on, 14–10 (II-A)

- individual protection of, 11–7 (II-A)
- max address size from, 11–3 (II-A)
- possible sizes for, 11–2 (II-A)
- size range of, 17–1 (II-B), 22–1 (II-C)
- virtual address space from, 11–2 (II-A)
- PAGES, CRB field for, 26–70 (III)
- pageSize, 15–2 (II-B), 20–2 (II-C)
- PALcode
 - access to kernel stack, 14–28 (II-A)
 - Alpha Linux support for, 24–9 (II-C)
 - argument registers used, F–37
 - barriers with, 5–22 (I)
 - CALL_PAL instruction, 4–135 (I)
 - compared to hardware instructions, 6–1 (I)
 - current defined, 27–5 (III)
 - debugging, F–89
 - event counters during debug, F–90
 - identifying the image, 27–5 (III)
 - illegal operand trap, 14–15 (II-A)
 - implementation-specific, 6–2 (I)
 - initial processor context for, F–92
 - initialization of, 27–4 (III)
 - initializing environment for, F–90
 - instead of microcode, 6–1 (I)
 - instruction format, 3–13 (I)
 - internal software registers, F–51
 - kernel activates, F–3
 - loading, 27–4 (III)
 - loading at multiprocessor boot, 27–28 (III)
 - memory management requirements, 11–3 (II-A)
 - new defined, 27–5 (III)
 - OpenVMS, defined for, 10–1 (II-A)
 - OS Loader and, F–3
 - overview, 6–1 (I)
 - processor state transitions, 14–34 (II-A)
 - queue data type support, 10–20 (II-A)
 - recognized instructions, 6–4 (I)
 - replacing, 6–3 (I)
 - required, 6–2 (I)
 - required instructions, 6–4 (I)
 - running environment, 6–2 (I)
 - special functions function support, 6–2 (I)
 - swapping currently executing, F–70
 - switching, 16–22 (II-B), 21–21 (II-C), 27–5 (III)
 - switching at multiprocessor boot, 27–28 (III)
 - Tru64 UNIX support for, 19–9 (II-B)
 - unexpected exceptions in, F–29
 - variants at loading, 27–4 (III)
 - variants at multiprocessor boot, 27–28 (III)
 - variants at processor initialization, 27–24 (III)
 - version control, F–10
- See also Queues, support for
- PALcode available, per-CPU slot field for, 26–20 (III)
- PALcode image base address (PAL_BASE) register, F–8
 - from initpal, F–48
 - previous, F–93
 - structure of, F–93
- PALcode instructions
 - Alpha Linux privileged (list), 21–9 (II-C)
 - Alpha Linux unprivileged (list), 21–1 (II-C)
 - opcodes and format summarized, C–1
 - OpenVMS (list), 10–1 (II-A)
 - OpenVMS privileged (list), 10–81 (II-A)
 - OpenVMS unprivileged (list), 10–3 (II-A)
 - required, C–24
 - required privileged, 6–4 (I)
 - required unprivileged, 6–4 (I)
 - reserved, function codes for, C–24
 - Tru64 UNIX privileged (list), 16–10 (II-B)
 - Tru64 UNIX unprivileged (list), 16–1 (II-B)
 - VAX compatibility, 10–73 (II-A)
 - Windows NT Alpha privileged (list), F–38
 - Windows NT Alpha unprivileged (list), F–81
- PALcode instructions, Alpha Linux privileged
 - cache flush, 21–10 (II-C)
 - console service, 21–11 (II-C)
 - performance monitoring function, 21–31 (II-C)
 - read machine check error summary, 21–12 (II-C)
 - read processor status, 21–13 (II-C)
 - read system value, 21–15 (II-C)
 - read user stack pointer, 21–14 (II-C)
 - return from system call, 21–16 (II-C)
 - return from trap, fault, or interrupt, 21–17 (II-C)
 - swap IPL, 21–20 (II-C)
 - swap PALcode image, 21–21 (II-C)
 - swap process context, 21–18 (II-C)
 - TB (translation buffer) invalidate, 21–23 (II-C)
 - wait for interrupt, 21–37 (II-C)
 - who am I, 21–24 (II-C)
 - write ASN, 21–25 (II-C)
 - write floating-point enable, 21–27 (II-C)
 - write interprocessor interrupt request, 21–28 (II-C)
 - write kernel global pointer, 21–29 (II-C)
 - write machine check error summary, 21–30 (II-C)
 - write system entry address, 21–26 (II-C)
 - write system page table base, 21–32 (II-C)
 - write system value, 21–34 (II-C)
 - write user stack pointer, 21–33 (II-C)
 - write virtual address boundary, 21–35 (II-C)

- write virtual page table pointer, 21–36 (II-C)
- PALcode instructions, Alpha Linux unprivileged
 - breakpoint, 21–2 (II-C)
 - bugcheck, 21–3 (II-C)
 - clear floating-point enable, 21–5 (II-C)
 - generate trap, 21–6 (II-C)
 - read unique value, 21–7 (II-C)
 - system call, 21–4 (II-C)
 - write unique value, 21–8 (II-C)
- PALcode instructions, OpenVMS privileged
 - cache flush, 10–82 (II-A)
 - console service, 10–83 (II-A)
 - load quadword physical, 10–84 (II-A)
 - move from processor register, 10–85 (II-A)
 - move to processor register, 10–86 (II-A)
 - store quadword physical, 10–87 (II-A)
 - swap PALcode image, 10–91 (II-A)
 - swap privileged context, 10–88 (II-A)
- PALcode instructions, OpenVMS unprivileged
 - breakpoint, 10–4 (II-A)
 - bugcheck, 10–5 (II-A)
 - change to executive mode, 10–6 (II-A)
 - change to kernel mode, 10–7 (II-A)
 - change to supervisor mode, 10–8 (II-A)
 - change to user mode, 10–9 (II-A)
 - clear floating-point trap, 10–10 (II-A)
 - generate software trap, 10–11 (II-A)
 - insert into queue (list), 10–28 (II-A)
 - probe for read access, 10–12 (II-A)
 - probe for write access, 10–12 (II-A)
 - read processor status, 10–13 (II-A)
 - read system cycle counter, 10–16 (II-A)
 - read unique context, 10–79 (II-A)
 - return from exception or interrupt, 10–14 (II-A)
 - swap AST enable, 10–18 (II-A)
 - thread, 10–78 (II-A)
 - write PS software field, 10–19 (II-A)
 - write unique context, 10–80 (II-A)
- PALcode instructions, Tru64 UNIX privileged
 - cache flush, 16–11 (II-B)
 - console service, 16–12 (II-B)
 - performance monitoring function, 16–32 (II-B)
 - read machine check error summary, 16–13 (II-B)
 - read processor status, 16–14 (II-B)
 - read system value, 16–16 (II-B)
 - read user stack pointer, 16–15 (II-B)
 - return from system call, 16–17 (II-B)
 - return from trap, fault, or interrupt, 16–18 (II-B)
 - swap IPL, 16–21 (II-B)
 - swap PALcode image, 16–22 (II-B)
 - swap process context, 16–19 (II-B)
- TB (translation buffer) invalidate, 16–24 (II-B)
- wait for interrupt, 16–38 (II-B)
- who am I, 16–25 (II-B)
- write ASN, 16–26 (II-B)
- write floating-point enable, 16–28 (II-B)
- write interprocessor interrupt request, 16–29 (II-B)
- write kernel global pointer, 16–30 (II-B)
- write machine check error summary, 16–31 (II-B)
- write system entry address, 16–27 (II-B)
- write system page table base, 16–33 (II-B)
- write system value, 16–35 (II-B)
- write user stack pointer, 16–34 (II-B)
- write virtual address boundary, 16–36 (II-B)
- write virtual page table pointer, 16–37 (II-B)
- PALcode instructions, Tru64 UNIX unprivileged
 - breakpoint, 16–2 (II-B)
 - bugcheck, 16–3 (II-B)
 - clear floating-point enable, 16–5 (II-B)
 - generate trap, 16–6 (II-B)
 - read unique value, 16–7 (II-B)
 - system call, 16–4 (II-B)
 - write unique value, 16–8 (II-B), 16–9 (II-B)
- PALcode instructions, Windows NT Alpha privileged
 - clear software interrupt request, F–40
 - data TB invalidate single, F–44
 - disable alignment fixups, F–41, F–45
 - disable all interrupts, F–42
 - drain all aborts, F–43
 - enable alignment fixups, F–45
 - enable interrupts, F–46
 - halt operating system, F–47
 - initialize PALcode data structures, F–48, F–50
 - initialize processor control region data, F–50
 - read current IRQL, F–52
 - read initial kernel stack pointer, F–53
 - read internal processor state, F–57
 - read machine check error summary register, F–54
 - read processor (PSR) status register, F–56
 - read processor control region base address, F–55
 - read software event counters, F–51
 - read thread value, F–58
 - restart operating system, F–60
 - return from exception or interrupt, F–63
 - return from system service call exception, F–61
 - set software interrupt request, F–65
 - swap current IRQL, F–68
 - swap current PALcode, F–70
 - swap initial kernel stack pointer, F–69
 - swap process context, F–71

- swap thread context, F-66
- transfer to console firmware, F-59
- translation buffer invalidate all, F-72
- translation buffer invalidate multiple, F-73
- translation buffer invalidate multiple for ASN, F-74
- translation buffer invalidate single, F-75
- translation buffer invalidate single for ASN, F-76
- write kernel exception entry routine, F-77
- write machine check error summary register, F-79
- write performance monitor, F-80
- PALcode instructions, Windows NT Alpha unprivileged
 - breakpoint trap, F-82
 - call kernel debugger, F-83
 - generate a trap, F-86
 - instruction memory barrier, F-87
 - kernel breakpoint trap, F-88
 - read TEB pointer, F-89
 - system service call, F-84
- PALcode loaded (PL) flag, 27-4 (III)
 - multiprocessor booting and, 27-27 (III)
 - per-CPU state contains, 26-22 (III)
- PALcode loading at bootstrap, 27-17 (III)
- PALcode memory space
 - length of, 26-16 (III)
 - PALcode loading and, 27-4 (III)
 - physical address of, 26-17 (III)
- PALcode memory valid (PMV) flag
 - multiprocessor booting and, 27-27 (III)
 - PALcode loading and, 27-4 (III)
 - per-CPU state contains, 26-22 (III)
- PALcode revision, per-CPU slot field for, 26-17 (III)
 - PALcode switching and, 27-6 (III)
- PALcode scratch space
 - length of, 26-17 (III)
 - PALcode loading and, 27-4 (III)
 - physical address of, 26-17 (III)
- PALcode scratch value
 - HWPCB, initial and, 27-25 (III)
- PALcode swapping, 10-91 (II-A)
- PALcode valid (PV) flag
 - multiprocessor booting and, 27-27 (III)
 - PALcode loading and, 27-4 (III)
 - per-CPU state contains, 26-22 (III)
- PALcode variation 2, 27-7 (III)
- PALcode variation assignments, D-4
- Panic exception (PANIC_ENTRY) register, F-8
- Panic exceptions, F-28
 - kernel stack under/overflow, F-89
 - trap from and dispatch for, F-29
- Panic stack, F-11
- Panic stack pointer, F-9
- PANIC_STACK_SWITCH code, F-29
- Passive release interrupts, 14-19 (II-A)
 - entry point, 14-27 (II-A)
- PASSIVE_LEVEL, IRQ table index name, F-5
- PC halted, per-CPU slot fields for, 26-18 (III)
- PC. See Program counter
- PCB. See Process control block
- PCBB. See Process control block base
- PCC_CNT, 3-3 (I), 4-145 (I)
- PCC_OFF, 3-3 (I), 4-145 (I)
- PCE bit, machine check error summary register, 13-14 (II-A), 19-8 (II-B), 24-8 (II-C), F-36
- Per-CPU slots
 - block for, 26-9 (III)
 - fields for, 26-16 (III)
 - HWRPB in, 26-14 (III)
 - number, HWRPB field for, 26-7 (III)
 - PALcode switching and, 27-7 (III)
 - size, HWRPB field for, 26-7 (III)
 - state flags at multiprocessor boot, 27-27 (III)
 - state flags in, 26-22 (III)
- Performance monitor interrupt entry point, 14-27 (II-A)
- Performance monitoring, E-7, E-12, E-22
- Performance monitoring enable (PME) bit
 - defined, 15-4 (II-B), 20-4 (II-C)
 - HWPCB and, 12-2 (II-A)
 - privileged context, 10-90 (II-A)
 - process context and, 18-1 (II-B), 23-1 (II-C)
- Performance monitoring register (PERFMON), 13-15 (II-A)
 - writing, 16-32 (II-B), 21-31 (II-C)
- Performance optimizations
 - branch prediction, A-3
 - code sequences, A-11
 - data stream, A-5
 - for I-streams, A-2
 - instruction alignment, A-2
 - instruction scheduling, A-5

- I-stream density, A-5
- multiple instruction issue, A-3
- shared data, A-6
- Performance tuning
 - IMPLVER instruction with, 4-141 (I)
- PERR (Pixel error) instruction, 4-157 (I)
- PFN
 - distributed memory cluster descriptor field, 27-15 (III)
 - null memory cluster descriptor field, 27-13 (III)
 - static memory cluster descriptor field, 27-12 (III)
 - See also Page frame number
- Physical address size, HWRPB field for, 26-6 (III)
- Physical address space, 11-3 (II-A), 17-3 (II-B), 22-3 (II-C), F-14
 - described, 5-1 (I)
- Physical address translation, 11-9 (II-A), 11-12 (II-A), 17-7 (II-B), 17-11 (II-B), 22-7 (II-C), 22-11 (II-C), F-14
- PHYSICAL_ADDRESS operator, 3-8 (I)
- Pipelined implementations, using EXCB instruction with, 4-138 (I)
- Pixel error instruction, 4-157 (I)
- PKLB (Pack longwords to bytes) instruction, 4-158 (I)
- PKWB (Pack words to bytes) instruction, 4-158 (I)
- PME. See Performance monitoring enable
- PMI bus, uncorrected protocol errors, 14-21 (II-A)
- Powerfail and recovery
 - multiprocessor type of, 27-33 (III)
 - split type of, 27-34 (III)
 - uniprocessor type of, 27-32 (III)
 - united type of, 27-33 (III)
- Powerfail interrupt, 14-20 (II-A)
 - service routine entry point, 14-27 (II-A)
- Powerfail restart (PR) flag
 - powerfail and recovery, 27-33 (III)
- POWERFAIL_RESTART, system variation field, 26-12 (III)
- Powerfail, CFLUSH PALcode instruction with, 14-20 (II-A)
- POWERFAIL, system variation field, 26-13 (III)
- Power-up initialization, 27-3 (III)
- Precise exceptions
 - VAX format arithmetic, 4-68 (I)
- Prefetch data (FETCHx instructions), 4-139 (I)
- PREFETCH instruction, 4-143 (I)
- Prefetch memory data (PREFETCHx instructions), 4-143 (I)
- PREFETCH_EN instruction, 4-143 (I)
- PREFETCH_M instruction, 4-143 (I)
 - data and locks with, 5-8 (I)
 - lock_flag with, 4-10 (I)
- PREFETCH_MEN instruction, 4-143 (I)
 - lock_flag with, 4-10 (I)
- Pre-PALcode initialization, F-91
- Primary bootstrap image
 - format of, 27-39 (III)
 - loading at cold, 27-18 (III)
- Primary processor
 - definition of, 25-1 (III)
 - modes for, 27-3 (III)
 - multiprocessor booting and, 27-27 (III)
 - running at multiprocessor boot, 27-29 (III)
 - switching from, 27-35 (III)
- Primary-eligible (PE) bit
 - BB_WATCH, 27-47 (III)
 - console switching and, 27-35 (III)
 - multiprocessor booting and, 27-27 (III)
- PRIORITY_ENCODE operator, 3-8 (I)
- PRIVATE_MCDS
 - null memory cluster descriptor field, 27-14 (III)
- Privileged Architecture Library. See PALcode
- Privileged context, 10-90 (II-A)
- Privileged context block base (PCBB) register, 13-16 (II-A)
 - PALcode switching and, 27-8 (III)
 - processor initialization and, 27-24 (III)
- Privileges, processor, F-6
- PROBER (PALcode) instruction, 10-12 (II-A)
- PROBEW (PALcode) instruction, 10-12 (II-A)
- Process, 12-1 (II-A)
 - context switching the, 12-4 (II-A)
- Process context, 18-1 (II-B), 23-1 (II-C)
 - saved in PCB, 18-2 (II-B), 23-2 (II-C)
- Process control block (PCB), 18-2 (II-B), 23-2 (II-C)
 - structure, 18-2 (II-B), 23-2 (II-C)
- Process control block (PCB) register, 15-3 (II-B),

- 20–3 (II-C)
- Process control block base (PCBB) register, 15–3 (II-B), 20–3 (II-C)
- Process control region base (PCR) register, F–8
- Process unique value (unique) register, 15–4 (II-B), 20–4 (II-C)
 - process context and, 18–1 (II-B), 23–1 (II-C)
- PROCESS_KEYCODE console terminal routine, 26–42 (III)
- Processor
 - adding to running system, 27–30 (III)
 - states and modes, 27–1 (III)
- Processor access modes, memory management, 11–7 (II-A)
- Processor available (PA) flag
 - multiprocessor booting and, 27–27 (III)
 - per-CPU state contains, 26–23 (III)
- Processor base (PRBR) register, 13–17 (II-A)
- Processor communication, 5–15 (I)
- Processor control block (PRCB)
 - initialization, F–92
- Processor control region, F–9
 - interrupt tables with, F–9
- Processor control region base (PCR) register
 - initialization, F–92
 - initializing, F–48
 - returning contents of, F–55
- Processor correctable errors, F–35
 - reporting, F–36
- Processor cycle counter (PCC) register, 3–3 (I)
 - Alpha Linux, 20–3 (II-C)
 - HWPCB, initial and, 27–25 (III)
 - OpenVMS, 9–2 (II-A)
 - RPCC instruction with, 4–145 (I)
 - system cycle counter with, 10–16 (II-A)
 - Tru64 UNIX, 15–3 (II-B)
 - See also Charged process cycles
- Processor data areas, F–9
- Processor hardware interrupt, service routine entry points, 14–27 (II-A)
- Processor initialization, 27–23 (III)
- Processor issue constraints, 5–13 (I)
- Processor issue sequence, 5–12 (I)
- Processor modes, 11–1 (II-A), 27–3 (III), F–5
 - AST pending state, 13–7 (II-A)
 - change to executive, 10–6 (II-A)
 - change to kernel, 10–7 (II-A)
 - change to supervisor, 10–8 (II-A)
 - change to user, 10–9 (II-A)
 - controlling memory access, 11–7 (II-A)
 - enabling reads, 11–4 (II-A)
 - enabling writes, 11–4 (II-A)
 - page access with, 11–2 (II-A)
 - PALcode state transitions, 14–34 (II-A)
- Processor number, reading, 13–31 (II-A)
- Processor present (PP) flag
 - multiprocessor booting and, 27–27 (III)
 - per-CPU state contains, 26–22 (III)
- Processor stacks, 14–7 (II-A)
- Processor state transitions, 14–34 (II-A)
- Processor state, defined, 14–4 (II-A)
- Processor state, internal, initialized, F–91
- Processor status (PS) register
 - bit meanings for, 19–2 (II-B), 24–2 (II-C)
 - bit summary, 14–5 (II-A)
 - bootstrap values in, 14–6 (II-A)
 - current, 14–5 (II-A)
 - defined, 9–1 (II-A), 15–4 (II-B), 20–4 (II-C)
 - explicit reading/writing of, 14–5 (II-A)
 - PALcode switching and, 27–8 (III)
 - process context and, 18–1 (II-B), 23–1 (II-C)
 - processor initialization and, 27–24 (III)
 - processor state and, 14–4 (II-A)
 - saved on stack, 14–5 (II-A)
 - saved on stack frame, 14–7 (II-A)
 - WR_PS_SW instruction, 10–19 (II-A)
- Processor status (PSR) register, F–5, F–8
 - returning contents of, F–56
- Processor type assignments, D–1
- Processor uncorrectable errors, F–35
- Processor unique value, 27–8 (III)
- Processor unique value (unique) register
 - HWPCB, initial and, 27–25 (III)
 - PALcode switching and, 27–8 (III)
- Processor, per-CPU slot field for
 - halt, 26–18 (III)
 - revision, 26–18 (III)
 - serial number, 26–18 (III)
 - software compatibility, 26–20 (III)
 - type, 26–17 (III)
 - variation, 26–18 (III)
- Processors, switching primary, 26–63 (III)
- Program counter (PC) register, 3–1 (I)
 - alignment, 14–6 (II-A)

- arithmetic traps and, 14–13 (II-A), 19–1 (II-B), 24–1 (II-C)
- current PC defined, 14–2 (II-A)
- defined, 15–3 (II-B), 20–3 (II-C)
- EXCB instruction and, 4–138 (I)
- explicit reading of, 14–6 (II-A)
- faults and, 14–8 (II-A)
- interrupts and, 14–2 (II-A)
- machine checks and, 14–21 (II-A)
- PALcode switching and, 27–8 (III)
- process context and, 18–1 (II-B), 23–1 (II-C)
- processor state and, 14–4 (II-A)
- saved on stack frame, 14–7 (II-A)
- synchronous traps and, 14–14 (II-A)

Program I/O mode, 27–3 (III)

Protection code, 11–7 (II-A), 17–7 (II-B), 22–7 (II-C)

Protection modes, 14–7 (II-A)

PS. See Processor status

PS<SP_ALIGN> field, 10–13 (II-A)

Pseudo-ops, A–16

PSR. See Processor status register

PSWITCH console routine, 26–63 (III), 27–36 (III)

PTBR. See Page table base

PTE. See Page table entry

PUTS console terminal routine, 26–36 (III)

Q

Quadword data type, 2–2 (I)

- alignment of, 2–3 (I), 2–11 (I)
- atomic access of, 5–2 (I)
- integer floating-point format, 2–11 (I)
- loading in physical memory, 10–84 (II-A)
- storing to physical memory, 10–87 (II-A)
- T_floating with, 2–11 (I)

Queues, support for

- absolute longword, 10–20 (II-A)
- absolute quadword, 10–23 (II-A)
- PALcode instructions (list), 10–28 (II-A)
- self-relative longword, 10–20 (II-A)
- self-relative quadword, 10–24 (II-A)

R

R31

- arithmetic traps and, 14–11 (II-A)
- destination register, 3–1 (I)
- restrictions, 3–1 (I)

- RAZ (read as zero), 1–8 (I)
- RC (read and clear) instruction, 4–153 (I)
- RD_PS (PALcode) instruction, 10–13 (II-A)
- rdcounters (PALcode) instruction, F–51
- rdirql (PALcode) instruction, F–52
- rdksp (PALcode) instruction, F–53
 - reads IKSP register, F–7
 - reads kernel stack, F–10
- rdmces (PALcode) instruction, 16–13 (II-B), 21–12 (II-C), F–54
- rdpcr (PALcode) instruction, F–55
 - reads PCR register, F–8
- rdps (PALcode) instruction, 16–14 (II-B), 21–13 (II-C)
- rdpsr (PALcode) instruction, F–56
- rdstate (PALcode) instruction, F–57
- rdteb (PALcode) instruction, F–89
 - reads TEB register, F–9
- rdthread (PALcode) instruction, F–58
 - reads THREAD register, F–9
- RDUNIQUE (PALcode) instruction
 - required recognition of, 6–4 (I)
- rdunique (PALcode) instruction, 16–7 (II-B), 21–7 (II-C)
- rdusp (PALcode) instruction, 16–15 (II-B), 21–14 (II-C)
- rdval (PALcode) instruction, 16–16 (II-B), 21–15 (II-C)
- READ device routine, 26–53 (III)
- Read/write ordering (multiprocessor), 5–10 (I)
 - determining requirements, 5–10 (I)
 - hardware implications for, 5–28 (I)
 - memory location defined, 5–11 (I)
- READ_UNQ (PALcode) instruction, 10–79 (II-A)
- Reason-for-halt code, power-up initialization, 27–4 (III)
- reboot (PALcode) instruction, F–59
 - operation of, F–92
 - tasks and sequence for, F–94
- Reduced page table (RPT) mode, 22–10 (II-C)
 - ACV fault with, 17–14 (II-B)
 - physical access for PTE, 11–12 (II-A), 17–10 (II-B), 22–10 (II-C)
 - requirements for, 11–11 (II-A), 17–10 (II-B),

- 22–10 (II-C)
- virtual access for PTE, 11–13 (II-A), 17–12 (II-B), 22–12 (II-C)
- Reduced page table mode, 11–11 (II-A), 17–10 (II-B)
- Regions in physical address space, 5–1 (I)
- Regions, bootstrap address space, 27–17 (III)
- Register mask, floating-point and integer, F–24
- Register write mask, with arithmetic traps, 14–13 (II-A)
- Registers, 3–1 (I)
 - Alpha Linux usage, 20–1 (II-C)
 - floating-point, 3–2 (I)
 - integer, 3–1 (I)
 - IPRs as, 13–1 (II-A)
 - lock, 3–2 (I)
 - memory prefetch, 3–3 (I)
 - OpenVMS usage of, 9–1 (II-A)
 - optional, 3–3 (I)
 - processor cycle counter, 3–3 (I)
 - program counter (PC), 3–1 (I)
 - Tru64 UNIX usage, 15–1 (II-B)
 - value when unused, 3–9 (I)
 - VAX compatibility, 3–3 (I)
 - Windows NT Alpha usage of, F–4
 - See also specific registers
- Register-to-register move, A–15
- REI (PALcode) instruction, 10–14 (II-A)
 - arithmetic traps, 14–9 (II-A)
 - faults, 14–8 (II-A)
 - interrupt arbitration, 14–33 (II-A)
 - interrupts, 14–2 (II-A)
 - machine checks, 14–22 (II-A)
 - synchronous traps, 14–14 (II-A)
- Relational Operators, 3–8 (I)
- Remove from queue PALcode instructions
 - longword, 10–69 (II-A)
 - longword at head interlocked, 10–49 (II-A)
 - longword at head interlocked resident, 10–52 (II-A)
 - longword at tail interlocked, 10–59 (II-A)
 - longword at tail interlocked resident, 10–62 (II-A)
 - quadword, 10–71 (II-A)
 - quadword at head interlocked, 10–54 (II-A)
 - quadword at head interlocked resident, 10–57 (II-A)
 - quadword at tail interlocked, 10–64 (II-A)
 - quadword at tail interlocked resident, 10–67 (II-A)
- REMQHIL (PALcode) instruction, 10–49 (II-A)
- REMQHILR (PALcode) instruction, 10–52 (II-A)
- REMQHIQ (PALcode) instruction, 10–54 (II-A)
- REMQHIQR (PALcode) instruction, 10–57 (II-A)
- REMQTIL (PALcode) instruction, 10–59 (II-A)
- REMQTILR (PALcode) instruction, 10–62 (II-A)
- REMQTIQ (PALcode) instruction, 10–64 (II-A)
- REMQTIQR (PALcode) instruction, 10–67 (II-A)
- REMQUEL (PALcode) instruction, 10–69 (II-A)
- REMQUEL/D (PALcode) instruction, 10–69 (II-A)
- REMQUEQ (PALcode) instruction, 10–71 (II-A)
- REMQUEQ/D (PALcode) instruction, 10–71 (II-A)
- Representable result, 4–65 (I)
- Reserved instructions, opcodes for, C–25
- Reserved operand, 4–65 (I)
- RESET_ENV variable routine, 26–59 (III)
- RESET_TERM console terminal routine, 26–38 (III)
- restart (PALcode) instruction, F–60
 - tasks and sequence for, F–94
- Restart block
 - with catastrophic errors, F–37
- Restart block pointer, F–9, F–92
- Restart execution address (RESTART_ADDRESS) register, F–8
 - PALcode exit and, F–37
- RESTART RTN VA, HWRPB field for, 26–9 (III)
- RESTART value, HWRPB field for, 26–9 (III)
- Restart-capable (RC) flag
 - failed bootstrap and, 27–21 (III)
 - multiprocessor booting and, 27–27 (III)
 - per-CPU state contains, 26–23 (III)
 - processor initialization and, 27–23 (III)
 - secondary console and, 27–30 (III)
 - state transitions and, 27–1 (III)
- RESTORE_TERM console routine, 27–37 (III), 27–39 (III)
- RESTORE_TERM RTN VA, HWRPB field for, 26–8 (III)
- RESTORE_TERM value, HWRPB field for, 26–9 (III)
- Result latency, A–5

RET instruction, 4–23 (I)

retsys (PALcode) instruction, 16–17 (II-B), 21–16 (II-C), F–61
 PS with, 19–2 (II-B), 24–2 (II-C)
 use of, F–21

Revision, HWRPB field for, 26–6 (III)

rfe (PALcode) instruction, F–63
 compared to retsys, F–61
 use of, F–21

RIGHT_SHIFT(x,y) operator, 3–8 (I)

ROM boot block structure, 27–44 (III)

ROM bootstrapping, 27–44 (III)

Rounding modes. See Floating-point rounding modes

RPCC (read processor cycle counter) instruction, 4–145 (I)
 RSCC instruction with, 10–17 (II-A)

RPT. See Reduced page table mode

RS (read and set) instruction, 4–153 (I)

RSCC (PALcode) instruction, 10–16 (II-A)
 RPCC instruction with, 10–17 (II-A)

rti (PALcode) instruction, 16–18 (II-B), 21–17 (II-C)
 exceptions with, 19–1 (II-B), 24–1 (II-C)
 PS with, 19–2 (II-B), 24–2 (II-C)

RX BUFFER, inter-console communications buffer field, 26–77 (III)

RX/TX extension block, 26–12 (III), 26–76 (III)
 offset in HWRPB, 26–10 (III)

RX/TX EXTENT
 mapping, 26–75 (III)
 system variation field, 26–12 (III)

RXLEN, inter-console communications buffer field, 26–77 (III)

RXRDY flag, 26–75 (III)
 mapping, 26–76 (III)
 multiprocessor booting and, 27–27 (III)

RXTX buffer area, 26–77 (III)
 per-CPU slot field for, 26–19 (III)

S

/S qualifier

arithmetic trap completion, 4–73 (I)
 compare instructions and, B–2
 floating-point control quadword and, B–4
 FPCR as control for, B–2

NaNs and invalid ops with, B–2
 software completion (SWC) and, 14–12 (II-A), 19–6 (II-B), 24–6 (II-C)
 underflow and denorm numbers with, B–2
 VAX trapping mode, 4–70 (I)

S_floating data type

alignment of, 2–8 (I)
 compared to F_floating, 2–7 (I)
 exceptions, 2–7 (I)
 mapping, 2–7 (I)
 MAX/MIN, 4–66 (I)
 NaN with T_floating convert, 4–89 (I)
 operations, 4–63 (I)
 unaligned data and, 14–26 (II-A)

S4ADDL instruction, 4–27 (I)

S4ADDQ instruction, 4–29 (I)

S4SUBL instruction, 4–39 (I)

S4SUBQ instruction, 4–41 (I)

S8ADDL instruction, 4–27 (I)

S8ADDQ instruction, 4–29 (I)

S8SUBL instruction, 4–39 (I)

S8SUBQ instruction, 4–41 (I)

SAVE_ENV variable routine, 26–61 (III)

SAVE_TERM console routine, 27–37 (III), 27–38 (III)

SAVE_TERM RTN VA, HWRPB field for, 26–8 (III)

SAVE_TERM value, HWRPB field for, 26–8 (III)

SBZ (should be zero), 1–8 (I)

SCC. See System cycle counter

SCE bit, machine check error summary register, 13–14 (II-A), 19–8 (II-B), 24–8 (II-C), F–36

Secondary processors

definition of, 25–1 (III)
 modes for, 27–3 (III)
 multiprocessor booting and, 27–28 (III)

Security holes, 1–6 (I)

UNPREDICTABLE results and, 1–8 (I)

Seg0

mapping of, 17–1 (II-B), 22–1 (II-C)
 virtual format, 17–2 (II-B)

Seg1

mapping of, 17–1 (II-B), 22–1 (II-C)
 virtual format, 17–2 (II-B)

- Self-relative longword queue, 10–20 (II-A)
- Self-relative quadword queue, 10–24 (II-A)
- Sequential read/write, A–9
- Serialization, MB instruction with, 4–142 (I)
- SET_ENV variable routine, 26–58 (III)
- SET_TERM_CTL terminal console routine, 26–41 (III)
- SET_TERM_INT console terminal routine, 26–39 (III)
- SEXT(x) operator, 3–8 (I)
- Shared data (multiprocessor), A–6
 - changed vs. updated datum, 5–6 (I)
- Shared data structures
 - atomic update, 5–7 (I)
 - memory barrier (MB) instruction with, 5–9 (I)
 - ordering considerations, 5–9 (I)
- Shared memory
 - accessing, 5–12 (I)
 - defined, 5–11 (I)
- SHARED_MCDS
 - null memory cluster descriptor field, 27–14 (III)
- Shift arithmetic instructions, 4–47 (I)
- Sign extend instructions, 4–61 (I)
- Single-precision floating-point, 4–63 (I)
- SLL instruction, 4–46 (I)
- Software (SW) field, in PS register, 14–6 (II-A)
- Software completion bit, exception summary register, 14–13 (II-A), 19–6 (II-B), 24–6 (II-C), F–25
- Software considerations, A–1
 - See also Performance optimizations
- Software exceptions, F–26
- Software interrupt request (SIRR) register, F–9
 - clearing, F–40
 - described, 13–20 (II-A)
 - format for, F–33
 - interrupt arbitration, 14–32 (II-A), 14–33 (II-A)
 - protocol for, 14–18 (II-A)
 - See also Software interrupts
- Software interrupt summary (SISR) register
 - described, 13–21 (II-A)
 - processor initialization and, 27–24 (III)
 - protocol for, 14–18 (II-A)
- Software interrupts, 14–17 (II-A)
 - asynchronous system traps (AST), 14–18 (II-A)
 - protocol between summary and request, 14–18 (II-A)
 - recording pending state of, 13–21 (II-A)
 - request (SIRR) register, 14–18 (II-A)
 - requesting, 13–20 (II-A), F–33
 - requests after exception handling, F–61, F–63
 - service routine entry points, 14–27 (II-A)
 - setting, F–65
 - summary (SISR) register, 14–17 (II-A)
 - supported levels of, 13–20 (II-A)
- Software page coloring caches, 26–21 (III)
- Software traps, generating, 10–11 (II-A)
- SP. See Stack pointer
- SQRTF instruction, 4–128 (I)
- SQRTG instruction, 4–128 (I)
- SQRTS instruction, 4–129 (I)
- SQRTT instruction, 4–129 (I)
- Square root instructions
 - IEEE, 4–129 (I)
 - VAX, 4–128 (I)
- SRA instruction, 4–47 (I)
- SRL instruction, 4–46 (I)
- ssir (PALcode) instruction, F–65
 - sets software interrupts, F–34
- Stack alignment, 14–29 (II-A)
- Stack alignment (SP_ALIGN), field in saved PS, 14–5 (II-A)
- Stack frames, 14–7 (II-A), 19–3 (II-B), 24–3 (II-C)
- Stack pointer (SP) register
 - defined, 9–1 (II-A), 15–4 (II-B), 20–4 (II-C)
 - linkage for, 15–1 (II-B), 20–1 (II-C)
- State flags, per-CPU slot field for, 26–16 (III)
- Static memory cluster descriptor, 27–10 (III), 27–11 (III)
- STATUS_ALPHA_ARITHMETIC code, F–24
- STATUS_ALPHA_GENTRAP code, F–26
- STATUS_BREAKPOINT code, F–27
- STATUS_DATATYPE_MISALIGNMENT code, F–25
- STATUS_ILLEGAL_INSTRUCTION code, F–25
- STATUS_INVALID_ADDRESS code, F–26
- STB instruction, 4–16 (I)
 - big-endian support with, 2–13 (I)
- STF instruction, 4–95 (I)

- big-endian support with, 2–13 (I)
 - unaligned data and, 14–26 (II-A)
- STG instruction, 4–96 (I)
 - unaligned data and, 14–26 (II-A)
- STL instruction, 4–16 (I)
 - big-endian support with, 2–13 (I)
 - unaligned data and, 14–26 (II-A)
- STL_C instruction, 4–13 (I)
 - big-endian support with, 2–13 (I)
 - guaranteed ordering with LDL_L, 4–15 (I)
 - LDx_L instruction and, 4–13 (I)
 - processor lock register/flag and, 4–14 (I)
 - unaligned data and, 14–26 (II-A)
- Storage, defined, 5–15 (I)
- Store instructions
 - emulation of, 4–2 (I)
 - FETCH instruction, 4–139 (I)
 - multiprocessor environment, 5–6 (I)
 - serialization, 4–142 (I)
 - store byte, 4–16 (I)
 - store longword, 4–16 (I)
 - store longword conditional, 4–13 (I)
 - store quadword, 4–16 (I)
 - store quadword conditional, 4–13 (I)
 - store word, 4–16 (I)
 - STQ_U, 4–18 (I)
 - unaligned data and, 14–26 (II-A)
 - See also Floating-point store instructions
- Store memory integer instructions, 4–4 (I)
- STORE_CONDITIONAL operator, 3–8 (I)
- Store-conditional, defined, 5–16 (I)
- STQ instruction, 4–16 (I)
 - unaligned data and, 14–26 (II-A)
- STQ_C instruction, 4–13 (I)
 - guaranteed ordering with LDQ_L, 4–15 (I)
 - LDx_L instruction and, 4–14 (I)
 - processor lock register/flag and, 4–14 (I)
 - unaligned data and, 14–26 (II-A)
- STQ_U instruction, 4–18 (I)
- STQP (PALcode) instruction, 10–87 (II-A)
- STS instruction, 4–97 (I)
 - big-endian support with, 2–13 (I)
 - FPCR and, 4–84 (I)
 - unaligned data and, 14–26 (II-A)
- STT instruction, 4–98 (I)
 - unaligned data and, 14–26 (II-A)
- STW instruction, 4–16 (I)
 - big-endian support with, 2–13 (I)
- /SU qualifier
 - floating-point control quadword and, B–4
 - FPCR as control for, B–2
 - IEEE trapping mode, 4–72 (I)
 - VAX trapping mode, 4–71 (I)
- SUBF instruction, 4–130 (I)
- SUBG instruction, 4–130 (I)
- SUBL instruction, 4–38 (I)
- SUBQ instruction, 4–40 (I)
- SUBS instruction, 4–131 (I)
- SUBT instruction, 4–131 (I)
- Subtract instructions
 - subtract longword, 4–38 (I)
 - subtract quadword, 4–40 (I)
 - subtract scaled longword, 4–39 (I)
 - subtract scaled quadword, 4–41 (I)
 - See also Floating-point operate
- /SUI qualifier
 - floating-point control quadword and, B–4
 - FPCR as control for, B–2
 - IEEE trapping mode, 4–73 (I)
- SUM bit. See Summary bit
- Summary bit, in FPCR, 4–81 (I)
- Superpage address space, F–13
- Supervisor read enable (SRE), bit in PTE, 11–4 (II-A)
- Supervisor stack pointer (SSP) register, 13–22 (II-A)
 - HWPCB and, 12–2 (II-A)
 - HWPCB, initial and, 27–25 (III)
 - internal processor register, 13–1 (II-A)
- Supervisor write enable (SWE), bit in PTE, 11–4 (II-A)
- /SV qualifier
 - floating-point control quadword and, B–4
 - FPCR as control for, B–2
 - IEEE trapping mode, 4–72 (I)
 - VAX trapping mode, 4–71 (I)
- /SVI qualifier
 - floating-point control quadword and, B–4
 - FPCR as control for, B–2
 - IEEE trapping mode, 4–73 (I)
- SWASTEN (PALcode) instruction, 10–18 (II-A)
 - ASTEN register and, 13–6 (II-A)
 - interrupt arbitration, 14–34 (II-A)
- SWC bit
 - exception summary parameter, 14–12 (II-A)

- exception summary register, 19–2 (II-B), 19–6 (II-B), 24–2 (II-C), 24–6 (II-C), F–25
- SWPCTX (PALcode) instruction, 10–88 (II-A)
 - ASTSR register and, 13–8 (II-A)
- swpctx (PALcode) instruction, 16–19 (II-B), 21–18 (II-C), F–66
 - ASNs with, 17–12 (II-B), 22–12 (II-C)
 - PCB with, 18–2 (II-B), 23–2 (II-C)
 - PDR register with, F–8
 - writes IKSP register, F–7
 - writes TEB register, F–9
 - writes THREAD register, F–9
- swpipl (PALcode) instruction, 16–21 (II-B), 21–20 (II-C)
 - PS with, 19–2 (II-B), 24–2 (II-C)
- swpirql (PALcode) instruction, F–68
 - as synchronization function, F–33
- swpksp (PALcode) instruction, F–69
 - reads kernel stack, F–10
 - writes IKSP register, F–7
- SWPPAL (PALcode) instruction, 10–91 (II-A)
 - PALcode switching and, 27–6 (III)
 - required recognition of, 6–4 (I)
- swppal (PALcode) instruction, 16–22 (II-B), 21–21 (II-C), F–70, F–95
 - firmware contributes, F–2
 - required recognition of, 6–4 (I)
- swpprocess (PALcode) instruction, F–71
 - writes PDR register, F–8
- Synchronization levels, interrupt, F–31
- Synchronous traps, 14–9 (II-A), 19–2 (II-B), 24–2 (II-C)
 - data alignment, 14–14 (II-A)
 - defined, 14–9 (II-A)
 - program counter (PC) value, 14–14 (II-A)
 - REI instruction with, 14–14 (II-A)
- System call entry (entSys) register, 15–3 (II-B), 19–4 (II-B), 19–9 (II-B), 20–3 (II-C), 24–4 (II-C), 24–9 (II-C)
- System control block (SCB)
 - arithmetic trap entry points, 14–25 (II-A)
 - fault entry points, 14–25 (II-A)
 - finding PFN, 13–19 (II-A)
 - memory management faults and, 11–16 (II-A)
 - saved on stack frame, 14–7 (II-A)
 - structure of, 14–24 (II-A)
- System control block base (SCBB) register, 13–19 (II-A)
 - specifies PFN, 14–24 (II-A)
- System correctable errors, F–35
 - reporting, F–36
- System crash, requesting, 27–35 (III)
- System cycle counter (SCC) register
 - processor initialization and, 27–24 (III)
 - reading, 10–16 (II-A)
- System entry addresses, 19–4 (II-B), 24–4 (II-C)
- System initialization, 27–3 (III)
- System page table base (SYSPTBR) register, 13–23 (II-A)
 - PTBR and, 11–11 (II-A)
 - using, 11–11 (II-A)
- System page table base (SYSPTR) register
 - reduced page table mode with, 11–12 (II-A), 17–11 (II-B), 22–11 (II-C)
- System restarts, 27–31 (III)
 - error halt and recovery, 27–34 (III)
 - forcing console I/O mode, 27–39 (III)
 - powerfail and recovery (multiprocessor), 27–33 (III)
 - powerfail and recovery (split), 27–34 (III)
 - powerfail and recovery (uniprocessor), 27–32 (III)
 - powerfail and recovery (united), 27–33 (III)
 - primary switching, 27–35 (III)
 - requesting a crash, 27–35 (III)
 - RESTORE_TERM routine, 27–37 (III), 27–39 (III)
 - restoring terminal state, 27–37 (III)
 - SAVE_TERM routine, 27–37 (III), 27–38 (III)
 - saving terminal state, 27–37 (III)
- System serial number, HWRPB field for, 26–6 (III)
- System service call exceptions, F–23
 - returning from, F–61
- System service exception address (SYSCALL_ENTRY) register, F–9
- System type specific (STS), system variation field, 26–12 (III)
- System uncorrectable errors, F–35
- System value (sysvalue) register, 15–4 (II-B), 20–4 (II-C)
 - PALcode switching and, 27–8 (III)
- System variation field (HWRPB)
 - bit summary, 26–12 (III)
- System, HWRPB field for
 - revision code, 26–7 (III), 26–11 (III)
 - serial number, 26–11 (III)

type, 26–6 (III), 26–12 (III)
variation, 26–7 (III), 26–12 (III)
Sysvalue. See System value

T

T_floating data type

alignment of, 2–9 (I)
exceptions, 2–8 (I)
format, 2–8 (I)
MAX/MIN, 4–66 (I)
NaN with S_floating convert, 4–89 (I)
unaligned data and, 14–26 (II-A)

Tape. See Magtape

TB hint offset, HWRPB field for, 26–7 (III)

TB miss MB (NOMB), PTE bit, 11–5 (II-A), 17–4
(II-B), 22–4 (II-C)

TB. See Translation buffer

TBB. See Translation buffer hint block

tbi (PALcode) instruction, 16–24 (II-B), 21–23
(II-C)

ASN with, 16–24 (II-B)
TBs with, 17–12 (II-B), 22–12 (II-C)

tbia (PALcode) instruction, F–17, F–72

tbim (PALcode) instruction, F–17, F–18, F–73

tbimasn (PALcode) instruction, F–18, F–74

tbis (PALcode) instruction, F–18, F–75

tbisasn (PALcode) instruction, F–18, F–76

Temporary PALcode registers, F–37

Terminal console

setting controls, 26–41 (III)

Terminals

setting interrupts for, 26–39 (III)

TEST(x,cond) operator, 3–9 (I)

TESTED_PAGES

distributed memory cluster descriptor field,
27–15 (III)
static memory cluster descriptor field, 27–12
(III)

Thread environment block base (TEB) register, F–9

context switching and, F–12, F–67
initializing, F–48
returning contents of, F–89

Thread unique value (THREAD) register, F–9

context switching and, F–12, F–67
initializing, F–48
returning contents of, F–58

Timeliness of location access, 5–17 (I)

Timer support, HAL interface fpr, F–3

Timing considerations, atomic sequences, A–17

Translation

physical, 17–7 (II-B), 22–7 (II-C)
virtual, 17–9 (II-B), 22–9 (II-C)

Translation buffer (TB), 17–12 (II-B), 22–12 (II-C)

ASNs with, 11–13 (II-A), 13–26 (II-A), 16–24
(II-B), 21–23 (II-C)
context switching and, F–12
fault on execute, 14–11 (II-A)
fault on read, 14–10 (II-A)
fault on write, 14–11 (II-A)
invalid PTEs and, 11–14 (II-A)
invalidate all, F–72
invalidate multiple, F–73
invalidate single, F–75
invalidate single data, F–44
management of, F–17
recursion in, F–18

Translation buffer check (TBCHK) register

described, 13–24 (II-A)
translation buffer and, 11–14 (II-A)

Translation buffer hint block (TBB), 26–9 (III),
26–13 (III)

Translation buffer invalidate all (TBIA) register

described, 13–25 (II-A)
translation buffer and, 11–14 (II-A)

Translation buffer invalidate all process (TBIAP)
register

described, 13–26 (II-A)
translation buffer and, 11–14 (II-A)

Translation buffer invalidate single (TBIS) register,
13–27 (II-A)

Translation buffer miss memory barrier (NOMB)

bit in PTE, 11–5 (II-A), 17–4 (II-B), 22–4
(II-C)

Translation not valid (TNV) fault, 11–16 (II-A),
14–10 (II-A), 17–13 (II-B), 22–13 (II-C),
F–22

service routine entry point, 14–25 (II-A)

Trap disable bits, 4–79 (I)

denormal operand exception, 4–82 (I)
division by zero, 4–82 (I)
DZED with DZE arithmetic trap, 4–78 (I)
DZED with INV arithmetic trap, 4–77 (I)
IEEE compliance and, B–3
inexact result, 4–81 (I)
invalid operation, 4–82 (I)

- overflow disable, 4–82 (I)
- unimplemented, 4–79 (I)
- Trap enable bits, B–4
- Trap frames and offsets, F–21
- Trap handler, with non-finite arithmetic operands, 4–74 (I)
- Trap handling, IEEE floating-point, B–6
- Trap modes
 - floating-point, 4–70 (I)
- Trap shadow, 19–2 (II-B), 24–2 (II-C)
 - defined for floating-point, 4–65 (I)
 - programming implications for, 5–29 (I)
 - rules for, 4–74 (I)
- TRAP_CAUSE_UNKNOWN code, F–29
- TRAPB (trap barrier) instruction, A–15
 - described, 4–147 (I)
 - FPCR and, 4–84 (I)
- Trapping modes, floating-point, 4–70 (I)
- Traps. See Arithmetic traps
- TrFir trap frame offset
 - from ExceptionPC address, F–23
- Trigger instruction, 19–2 (II-B), 24–2 (II-C)
- Tru64 UNIX PALcode, instruction summary, C–19
- True result, 4–65 (I)
- True zero, 4–66 (I)
- TTY_DEV environment variable, 26–28 (III)
 - CTB and, 26–73 (III)
- TX BUFFER, inter-console communications buffer field, 26–77 (III)
- TXLEN, inter-console communications buffer field, 26–77 (III)
- TXRDY flag, 26–75 (III)
 - mapping, 26–76 (III)
 - multiprocessor booting and, 27–27 (III)

U

- /U qualifier
 - IEEE trapping mode, 4–72 (I)
 - VAX trapping mode, 4–70 (I)
- UMULH instruction, 4–37 (I)
 - MULQ and, 4–36 (I)
- Unaligned access exceptions, F–25
- Unaligned access fault
 - system entry for, 19–4 (II-B), 24–4 (II-C)

- UNALIGNED data objects, 1–8 (I)
- Unaligned fault entry (entUna) register, 15–3 (II-B), 19–9 (II-B), 20–3 (II-C), 24–9 (II-C)
- Unconditional long jump, 4–24 (I)
- UNDEFINED operations, 1–7 (I)
- Underflow bit, exception summary register, F–25
- Underflow enable (UNFE)
 - FP_C quadword bit, B–5
- Underflow status (UNFS)
 - FP_C quadword bit, B–5
- Underflow trap, 14–13 (II-A), 19–5 (II-B), 24–5 (II-C), F–25
- UNF bit
 - exception summary parameter, 14–12 (II-A)
 - exception summary register, 19–5 (II-B), 24–5 (II-C), F–25
- Unique
 - process unique value, 15–4 (II-B), 20–4 (II-C)
 - See also Processor unique value
- UNOP code form, A–13
- UNORDERED memory references, 5–10 (I)
- Unpack to bytes instructions, 4–159 (I)
- UNPKBL (Unpack bytes to longwords) instruction, 4–159 (I)
- UNPKBW (Unpack bytes to words) instruction, 4–159 (I)
- UNPREDICTABLE results, 1–7 (I)
- Updated datum, 5–6 (I)
- USAGE
 - distributed memory cluster descriptor field, 27–16 (III)
 - static memory cluster descriptor field, 27–12 (III)
- User read enable (URE)
 - bit in PTE, 11–4 (II-A), 17–4 (II-B), 22–4 (II-C)
- User stack, F–11
- User stack pointer (USP) register, 13–28 (II-A)
 - defined, 15–4 (II-B), 20–4 (II-C)
 - HWPCB and, 12–2 (II-A)
 - HWPCB, initial and, 27–25 (III)
 - internal processor register, 13–1 (II-A)
 - process context and, 18–1 (II-B), 23–1 (II-C)
- User write enable (UWE)
 - bit in PTE, 11–4 (II-A), 17–4 (II-B), 22–4 (II-C)

USER_BREAKPOINT breakpoint type, F-27
USP. See User stack pointer

V

/V qualifier

- IEEE trapping mode, 4-72 (I)
- VAX trapping mode, 4-70 (I)

Valid (V)

- bit in PTE, 11-6 (II-A), 17-6 (II-B), 22-6 (II-C), F-17

Validation, HWRPB field for, 26-6 (III)

vaSize, 15-2 (II-B), 20-2 (II-C)

VAX compatibility instructions, restrictions for, 4-152 (I)

VAX compatibility register, 3-3 (I)

VAX floating-point

- computational models, 4-68 (I)
- D_floating, 2-5 (I)
- F_floating, 2-3 (I)
- G_floating, 2-4 (I)
- high-performance arithmetic, 4-68 (I)
- reserved operand, 4-65 (I)
- See also Floating-point instructions

VAX floating-point instructions

- add, 4-109 (I)
- compare, 4-111 (I)
- convert from integer, 4-114 (I)
- convert to integer, 4-113 (I)
- convert VAX floating format, 4-115 (I)
- divide, 4-120 (I)
- function codes for, C-9
- function field format, 4-87 (I)
- integer move, from, 4-124 (I)
- multiply, 4-126 (I)
- operate, 4-101 (I)
- square root instructions, 4-128 (I)
- subtract, 4-130 (I)

VAX rounding modes, 4-67 (I)

VAX trapping modes, 4-70 (I)

- /S, 4-70 (I)
- /SU, 4-71 (I)
- /SV, 4-71 (I)
- /U, 4-70 (I)
- /V, 4-70 (I)
- default mode, 4-70 (I)
- precise, 4-70 (I)
- summary, 4-71 (I)

Vector instructions

byte and word maximum, 4-155 (I)

byte and word minimum, 4-155 (I)

Virtual address boundary (VIRBND) register, 13-29 (II-A)

- PALcode switching and, 27-8 (III)
- reduced page table mode with, 11-12 (II-A), 17-11 (II-B), 22-11 (II-C)
- support for, 26-12 (III)
- using, 11-11 (II-A)

Virtual address format, 11-2 (II-A)

Virtual address space, 11-1 (II-A), 11-2 (II-A), 17-1 (II-B), 22-1 (II-C), F-13

- minimum and maximum, 11-2 (II-A)
- page size with, 11-2 (II-A)

Virtual address translation, 11-10 (II-A), 11-13 (II-A), 17-9 (II-B), 17-12 (II-B), 22-9 (II-C), 22-12 (II-C), E-6, F-14

Virtual addresses

- format of, F-14
- non-canonical at fault, F-26
- physical view of, F-15
- virtual view of, F-14

Virtual cache blocks

- invalidating all, F-72
- invalidating multiple, F-73
- invalidating single, F-75

Virtual D-cache, 5-4 (I)

Virtual format, 22-2 (II-C)

Virtual I-cache, 5-4 (I)

- maintaining coherency of, 5-5 (I)

Virtual machine monitor (VMM), bit in PS register, 14-5 (II-A)

Virtual memory regions, initial, 27-19 (III)

Virtual page table base (VPTB)

- HWRPB field for, 26-7 (III)
- PALcode switching and, 27-7 (III)

Virtual page table base (VPTB) register, 13-30 (II-A)

Virtual page table pointer (VPTPTR), 15-5 (II-B), 20-5 (II-C)

Visibility, defined, 5-14 (I)

VPTB. See Virtual page table base

VPTPTR. See Virtual page table pointer

W

Waivers, E-1

Warm bootstrapping, 27-25 (III)

Watchpoints

 fault on read, 14-10 (II-A)

 fault on write, 14-11 (II-A)

WH64 instruction, 4-148 (I), A-11

 lock_flag with, 4-10 (I)

WH64EN instruction, 4-148 (I)

 lock_flag with, 4-10 (I)

whami (PALcode) instruction, 16-25 (II-B), 21-24 (II-C)

whami, current processor number, 15-5 (II-B), 20-5 (II-C)

Who-Am-I (WHAMI) register, 13-31 (II-A)

 PALcode switching and, 27-8 (III)

 processor initialization and, 27-24 (III)

WMB (Write memory barrier) instruction, 4-150 (I)

 atomic operations with, 5-8 (I)

 MB compared to, 4-151 (I)

 shared data structures and, 5-10 (I)

Word data type, 2-1 (I)

 atomic access of, 5-3 (I)

WR_PS_SW (PALcode) instruction, 10-19 (II-A)

wrasn (PALcode) instruction, 16-26 (II-B), 21-25 (II-C)

wrent (PALcode) instruction, 16-27 (II-B), 21-26 (II-C)

wrenry (PALcode) instruction, F-77

 initialization and, F-92

 writes GENERAL_ENTRY register, F-7

 writes INTERRUPT_ENTRY register, F-7

 writes MEM_MGMT_ENTRY register, F-8

 writes PANIC_ENTRY register, F-8

 writes SYSCALL_ENTRY register, F-9

wrfen (PALcode) instruction, 16-28 (II-B), 21-27 (II-C)

wripir (PALcode) instruction, 16-29 (II-B), 21-28 (II-C)

Write buffers, requirements for, 5-4 (I)

WRITE device routine, 26-55 (III)

 characteristics determined by OPEN, 26-56 (III)

WRITE_UNQ (PALcode) instruction, 10-80 (II-A)

Write-back caches, requirements for, 5-4 (I)

wrkgp (PALcode) instruction, 16-30 (II-B), 21-29 (II-C)

wrmces (PALcode) instruction, 16-31 (II-B), 21-30 (II-C), F-79

wrperfmon (PALcode) instruction, 16-32 (II-B), 21-31 (II-C), F-80

wrsysptb (PALcode) instruction, 16-33 (II-B), 21-32 (II-C)

wrunique (PALcode) instruction, 16-8 (II-B), 16-9 (II-B), 21-8 (II-C)

 required recognition of, 6-4 (I)

wrusp (PALcode) instruction, 16-34 (II-B), 21-33 (II-C)

wrval (PALcode) instruction, 16-35 (II-B), 21-34 (II-C)

wrvirbnd (PALcode) instruction, 16-36 (II-B), 21-35 (II-C)

wrvptpr (PALcode) instruction, 16-37 (II-B), 21-36 (II-C)

WTINT (PALcode) instruction, 10-93 (II-A)

wtint (PALcode) instruction, 16-38 (II-B), 21-37 (II-C)

X

x MOD y operator, 3-8 (I)

X_floating data type, 2-9 (I)

 alignment of, 2-9 (I)

 big-endian format, 2-10 (I)

 MAX/MIN, 4-66 (I)

XOR instruction, 4-43 (I)

XOR operator, 3-9 (I)

Y

YUV coordinates, interleaved, 4-154 (I)

Z

ZAP instruction, 4-62 (I)

ZAPNOT instruction, 4-62 (I)

Zero byte instructions, 4-62 (I)

ZEXT(x)operator, 3-9 (I)

Instruction Index

Index entries are keyed with the following suffixes:

Suffix	Location
(I)	Common Architecture
(II-A)	OpenVMS PALcode
(II-B)	Tru64 UNIX PALcode
(II-C)	Alpha Linux PALcode
F-	Windows NT Alpha

A

ADDF 4-109 (I)
ADDG 4-109 (I)
ADDL 4-26 (I)
ADDQ 4-28 (I)
ADDS 4-110 (I)
ADDT 4-110 (I)
AMASK 4-133 (I)
AMOVRM 10-74 (II-A)
AMOVRR 10-74 (II-A)
AND 4-43 (I)

B

BEQ 4-21 (I)
BGE 4-21 (I)
BGT 4-21 (I)
BIC 4-43 (I)
BIS 4-43 (I)
BLBC 4-21 (I)
BLBS 4-21 (I)
BLE 4-21 (I)
BLT 4-21 (I)
BNE 4-21 (I)
BPT 10-4 (II-A)
bpt 16-2 (II-B), 21-2 (II-C), F-82
BR 4-22 (I)

BSR 4-22 (I)
BUGCHK 10-5 (II-A)
bugchk 16-3 (II-B), 21-3 (II-C)

C

CALL_PAL 4-135 (I)
callkd F-83
callsys 16-4 (II-B), 21-4 (II-C), F-84
CFLUSH 10-82 (II-A)
cflush 16-11 (II-B), 21-10 (II-C)
CHME 10-6 (II-A)
CHMK 10-7 (II-A)
CHMS 10-8 (II-A)
CHMU 10-9 (II-A)
CLRFEN 10-10 (II-A)
clrfen 16-5 (II-B), 21-5 (II-C)
CMOVEQ 4-44 (I)
CMOVGE 4-44 (I)
CMOVGT 4-44 (I)
CMOVLBC 4-44 (I)
CMOVLBS 4-44 (I)
CMOVLE 4-44 (I)
CMOVNE 4-44 (I)
CMPBGE 4-50 (I)
CMPEQ 4-30 (I)
CMPGEQ 4-111 (I)
CMPGLE 4-111 (I)

CMPGLT 4-111 (I)
CMPLE 4-30 (I)
CMPLT 4-30 (I)
CMPTEQ 4-112 (I)
CMPTLE 4-112 (I)
CMPTLT 4-112 (I)
CMPTUN 4-112 (I)
CMPULE 4-31 (I)
CMPULT 4-31 (I)
CPYS 4-104 (I)
CPYSE 4-104 (I)
CPYSN 4-104 (I)
CSERVE 10-83 (II-A)
cserve 16-12 (II-B), 21-11 (II-C)
csir F-40
CTLZ 4-32 (I)
CTPOP 4-33 (I)
CTTZ 4-34 (I)
CVTDG 4-115 (I)
CVTGD 4-115 (I)
CVTGF 4-115 (I)
CVTGQ 4-113 (I)
CVTLQ 4-105 (I)
CVTQF 4-114 (I)
CVTQG 4-114 (I)
CVTQL 4-105 (I)
CVTQS 4-117 (I)
CVTQT 4-117 (I)
CVTST 4-118 (I)
CVTTQ 4-116 (I)
CVTTS 4-119 (I)

D

dalnfix F-41
di F-42
DIVF 4-120 (I)
DIVG 4-120 (I)
DIVS 4-121 (I)
DIVT 4-121 (I)
draina F-43
dtbis F-44

E

ealnfix F-45
ECB 4-136 (I)

ei F-46
EQV 4-43 (I)
EXCB 4-138 (I)
EXTBL 4-52 (I)
EXTLH 4-52 (I)
EXTLL 4-52 (I)
EXTQH 4-52 (I)
EXTQL 4-52 (I)
EXTWH 4-52 (I)
EXTWL 4-52 (I)

F

FBEQ 4-100 (I)
FBGE 4-100 (I)
FBGT 4-100 (I)
FBLE 4-100 (I)
FBLT 4-100 (I)
FBNE 4-100 (I)
FCMOVEQ 4-106 (I)
FCMOVGE 4-106 (I)
FCMOVGT 4-106 (I)
FCMOVLE 4-106 (I)
FCMOVLT 4-106 (I)
FCMOVNE 4-106 (I)
FETCH 4-139 (I)
FETCH_M 4-139 (I)
FTOIS 4-122 (I)
FTOIT 4-122 (I)

G

GENTRAP 10-11 (II-A)
gentrap 16-6 (II-B), 21-6 (II-C), F-86

H

halt F-47

I

imb F-87
IMPLVER 4-141 (I)
initpal F-48
initpcr F-50
INSBL 4-56 (I)
INSLH 4-56 (I)

INSL 4–56 (I)
INSQH 4–56 (I)
INSQHIL 10–29 (II-A)
INSQHILR 10–31 (II-A)
INSQHIQ 10–33 (II-A)
INSQHIQR 10–35 (II-A)
INSQL 4–56 (I)
INSQTIL 10–37 (II-A)
INSQTILR 10–39 (II-A)
INSQTIQ 10–41 (II-A)
INSQTIQR 10–43 (II-A)
INSQUEL 10–45 (II-A)
INSQUEQ 10–47 (II-A)
INSWH 4–56 (I)
INSWL 4–56 (I)
ITOFF 4–124 (I)
ITOFS 4–124 (I)
ITOFT 4–124 (I)

J

JMP 4–23 (I)
JSR 4–23 (I)
JSW_COROUTINE 4–23 (I)

K

kbpt F–88

L

LD_L 4–9 (I)
LDA 4–5 (I)
LDAH 4–5 (I)
LDBU 4–6 (I)
LDF 4–91 (I)
LDG 4–92 (I)
LDL 4–6 (I)
LDQ 4–6 (I)
LDQ_L 4–9 (I)
LDQ_U 4–8 (I)
LDQP 10–84 (II-A)
LDS 4–93 (I)
LDT 4–94 (I)
LDWU 4–6 (I)

M

MAXSB8 4–155 (I)
MAXSW4 4–155 (I)
MAXUB8 4–155 (I)
MAXUW4 4–155 (I)
MB 4–142 (I)
MF_FPCR 4–108 (I)
MFPR_IPR_name 10–85 (II-A)
MINSB8 4–155 (I)
MINSW4 4–155 (I)
MINUB8 4–155 (I)
MINUW4 4–155 (I)
MSKBL 4–58 (I)
MSKLB 4–58 (I)
MSKLL 4–58 (I)
MSKQH 4–58 (I)
MSKQL 4–58 (I)
MSKWH 4–58 (I)
MSKWL 4–58 (I)
MT_FPCR 4–108 (I)
MTPR_IPR_name 10–86 (II-A)
MULF 4–126 (I)
MULG 4–126 (I)
MULL 4–35 (I)
MULQ 4–36 (I)
MULS 4–127 (I)
MULT 4–127 (I)

O

ORNOT 4–43 (I)

P

PERR 4–157 (I)
PKLB 4–158 (I)
PKLW 4–158 (I)
PREFETCH 4–143 (I)
PREFETCH_EN 4–143 (I)
PREFETCH_M 4–143 (I)
PRFETCH_MEN 4–143 (I)
PROBE 10–12 (II-A)

R

RC 4–153 (I)
RD_PS 10–13 (II-A)

rdcounters F-51
rdirql F-52
rdksp F-53
rdmces 16-13 (II-B), 21-12 (II-C), F-54
rdpcr F-55
rdps 16-14 (II-B), 21-13 (II-C)
rdpsr F-56
rdstate F-57
rdteb F-89
rdthread F-58
rdunique 16-7 (II-B), 21-7 (II-C)
rdusp 16-15 (II-B), 21-14 (II-C)
rdval 16-16 (II-B), 21-15 (II-C)
READ_UNQ 10-79 (II-A)
reboot F-59
REI 10-14 (II-A)
REMQHIL 10-49 (II-A)
REMQHILR 10-52 (II-A)
REMQHIQ 10-54 (II-A)
REMQHIQR 10-57 (II-A)
REMQTIL 10-59 (II-A)
REMQTILR 10-62 (II-A)
REMQTIQ 10-64 (II-A)
REMQTIQR 10-67 (II-A)
REMQUEL 10-69 (II-A)
REMQUEQ 10-71 (II-A)
restart F-60
RET 4-23 (I)
retsys 16-17 (II-B), 21-16 (II-C), F-61
rfe F-63
RPCC 4-145 (I)
RS 4-153 (I)
RSCC 10-16 (II-A)
rti 16-18 (II-B), 21-17 (II-C)

S

S4ADDL 4-27 (I)
S4ADDQ 4-29 (I)
S4SUBL 4-39 (I)
S4SUBQ 4-41 (I)
S8ADDL 4-27 (I)
S8ADDQ 4-29 (I)
S8SUBL 4-39 (I)
S8SUBQ 4-41 (I)
SEXTB 4-61 (I)

SEXTW 4-61 (I)
SLL 4-46 (I)
SQRTF 4-128 (I)
SQRTG 4-128 (I)
SQRTS 4-129 (I)
SQRTT 4-129 (I)
SRA 4-47 (I)
SRL 4-46 (I)
ssir F-65
STB 4-16 (I)
STF 4-95 (I)
STG 4-96 (I)
STL 4-16 (I)
STL_C 4-13 (I)
STQ 4-16 (I)
STQ_C 4-13 (I)
STQ_U 4-18 (I)
STQP 10-87 (II-A)
STS 4-97 (I)
STT 4-98 (I)
STW 4-16 (I)
SUBF 4-130 (I)
SUBG 4-130 (I)
SUBL 4-38 (I)
SUBQ 4-40 (I)
SUBS 4-131 (I)
SUBT 4-131 (I)
SWASTEN 10-18 (II-A)
swpctx 16-19 (II-B), 21-18 (II-C), F-66
swpipl 16-21 (II-B), 21-20 (II-C)
swpirql F-68
swpksp F-69
SWPPAL 10-91 (II-A)
swppal 16-22 (II-B), 21-21 (II-C), F-70
swpprocess F-71

T

tbi 16-24 (II-B), 21-23 (II-C)
tbia F-72
tbim F-73
tbimasn F-74
tbis F-75
tbisasn F-76
TRAPB 4-147 (I)

U

UMULH 4–37 (I)
UNPKBL 4–159 (I)
UNPKBW 4–159 (I)
urti 16–8 (II-B)

W

WH64 4–148 (I)
WH64EN 4–148 (I)
whami 16–25 (II-B), 21–24 (II-C)
WMB 4–150 (I)
WR_PS_SW 10–19 (II-A)
wrasn 16–26 (II-B), 21–25 (II-C)
wrent 16–27 (II-B), 21–26 (II-C)
wrenry F–77
wrfen 16–28 (II-B), 21–27 (II-C)
wripir 16–29 (II-B), 21–28 (II-C)
WRITE_UNQ 10–80 (II-A)
wrkgp 16–30 (II-B), 21–29 (II-C)
wrnces 16–31 (II-B), 21–30 (II-C), F–79
wrperfmon 16–32 (II-B), 21–31 (II-C), F–80
wrsysptb 16–33 (II-B), 21–32 (II-C)
wrunique 16–9 (II-B), 21–8 (II-C)
wrusp 16–34 (II-B), 21–33 (II-C)
wrval 16–35 (II-B), 21–34 (II-C)
wrvirbnd 16–36 (II-B), 21–35 (II-C)
wrvtpttr 16–37 (II-B), 21–36 (II-C)
WTINT 10–93 (II-A)
wtint 16–38 (II-B), 21–37 (II-C)

X

XOR 4–43 (I)

Z

ZAP 4–62 (I)
ZAPNOT 4–62 (I)

