COMPAQ

Alpha Architecture Handbook

Order Number: EC–QD2KC–TE

Revision/Update Information: This is Version 4 of the Alpha Architecture Handbook.

Compaq Computer Corporation

October 1998

Table of Contents

1 Introduction

2 Basic Architecture

3 Instruction Formats

4 Instruction Descriptions

5 System Architecture and Programming Implications

6 Common PALcode Architecture

B IEEE Floating-Point Conformance

C Instruction Summary

D Registered System and Processor Identifiers

E Waivers and Implementation-Dependent Functionality

Index

Figures

Tables

Preface

Chapters 1 through 8 and appendixes A through E of this book are directly derived from the *Alpha System Reference Manual, Version 7* and passed engineering change orders (ECOs) that have been applied. It is an accurate representation of the described parts of the Alpha architecture.

References in this handbook to the *Alpha Architecture Reference Manual* are to the Third Edition of that manual, EY-W938E-DP.

Chapter 1

Introduction

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

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

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

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

1.1 The Alpha Approach to RISC Architecture

Alpha Is a True 64-Bit Architecture

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

Alpha Is Designed for Very High-Speed Implementations

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

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

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

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

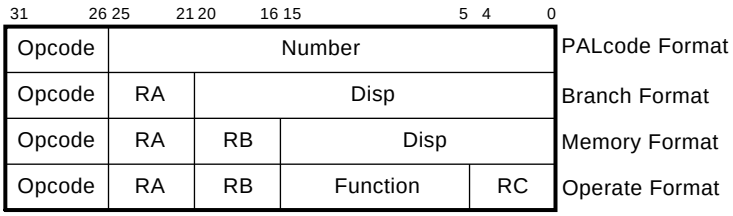
Alpha is a load/store RISC architecture with the following data characteristics:

- 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 operates preserve these canonical forms.

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

Integer Operate Instructions

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

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

There is no integer divide instruction.

The Alpha architecture also supports the following additional operations:

- Scaled add/subtract instructions for quick subscript calculation
- 128-bit multiply for division by a constant, and multiprecision arithmetic
- Conditional move instructions for avoiding branch instructions
- An extensive set of in-register byte and word manipulation instructions
- 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 are stylized code forms found in Section A.4.6.

Chapter 2

Basic Architecture

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.

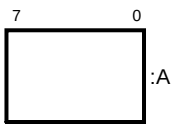
2.2 Data Types

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

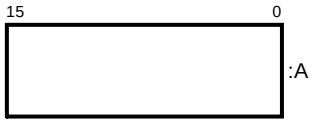


A byte is specified by its address A. A byte is an 8-bit value. The byte is only supported in Alpha by the load, store, sign-extend, extract, mask, insert, and zap instructions.

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



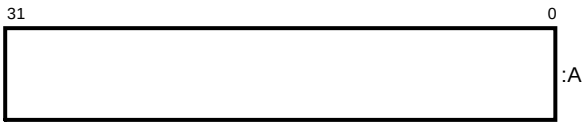
A word is specified by its address, the address of the byte containing bit 0.

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.

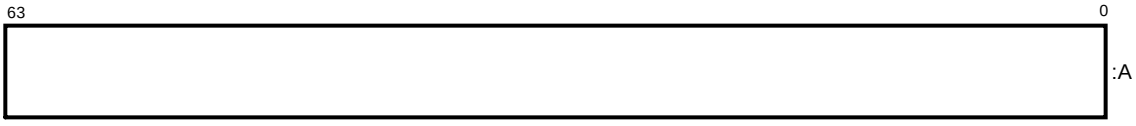
Note:

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.)

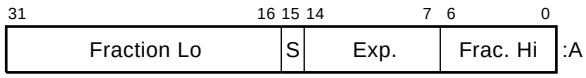
2.2.5 VAX Floating-Point Formats

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

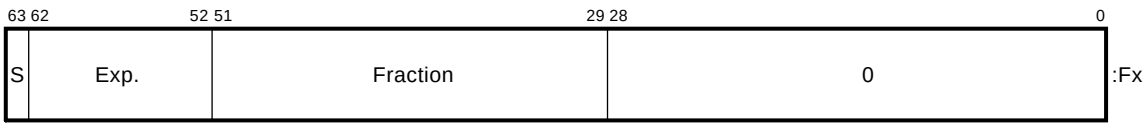
An F_floating datum is 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2–5 .

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.

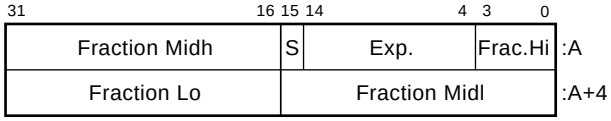
Note:

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

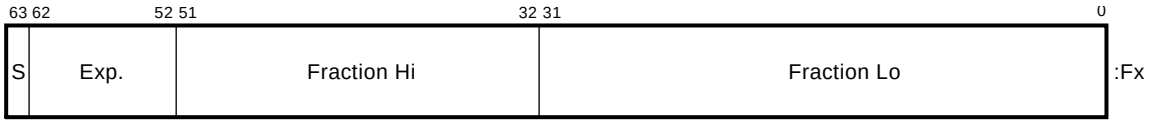
A G_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–7.

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*1 0**–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.

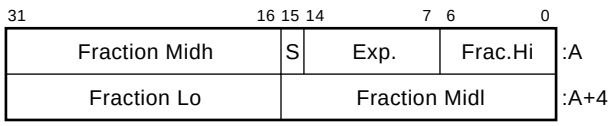
Note:

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

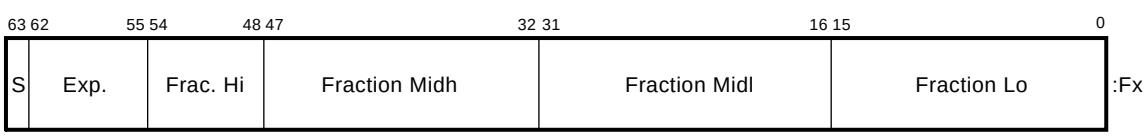
A D_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–9.

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 x 2**E x 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

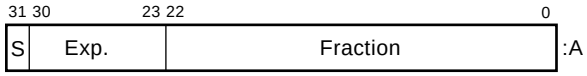
NaNs. Signaling NaNs are used to provide values for uninitialized variables and for arithmetic enhancements. Quiet NaNs provide retrospective diagnostic information regarding previous invalid or unavailable data and results. Signaling NaNs signal an invalid operation when they are an operand to an arithmetic 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

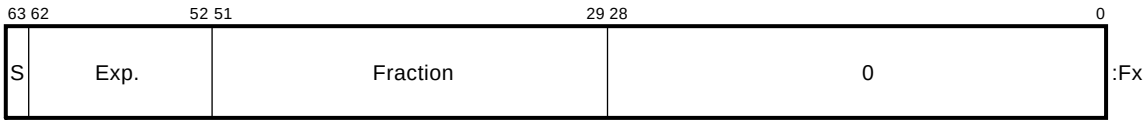
An IEEE single-precision, or S_floating, datum occupies 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2–11.

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.

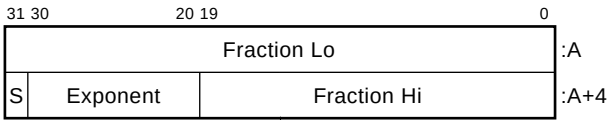
Note:

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

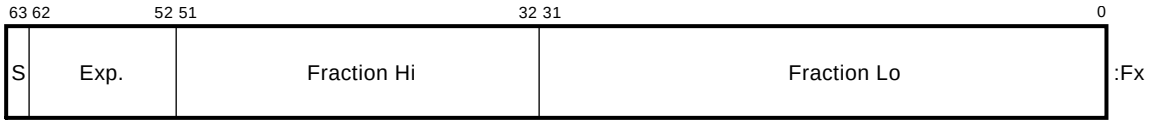
An IEEE double-precision, or T_floating, datum occupies 8 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–13.

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:

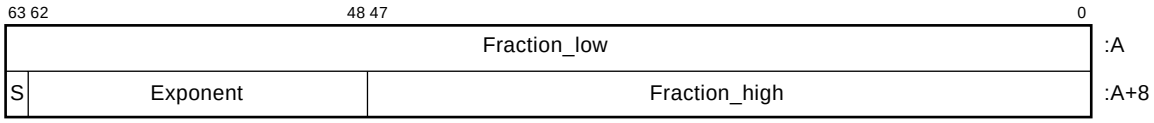
Alpha implementations will impose a significant performance penalty when accessing T_floating operands that are not naturally aligned. (A naturally aligned T_floating datum has zero as the low-order three bits of its address.)

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.

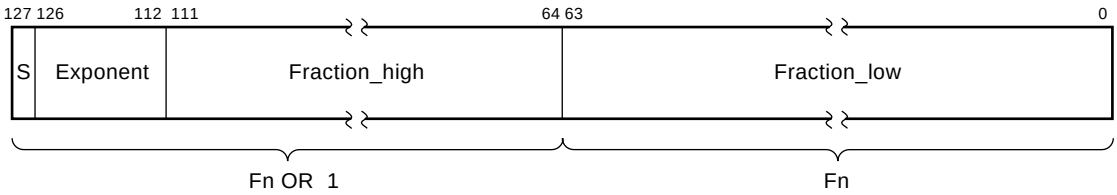
An IEEE extended-precision, or X_floating, datum occupies 16 contiguous bytes in memory, starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 127, as shown in Figure 2–15.

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

Note:

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

X_Floating Big-Endian Formats

Section 2.3 describes Alpha support for big-endian data types. It is intended that software or hardware implementation for a big-endian X_float data type comply with that support and have the following formats.

Figure 2–17: X_floating Big-Endian Datum

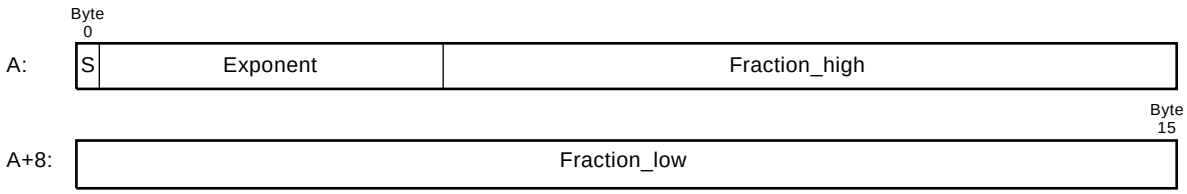
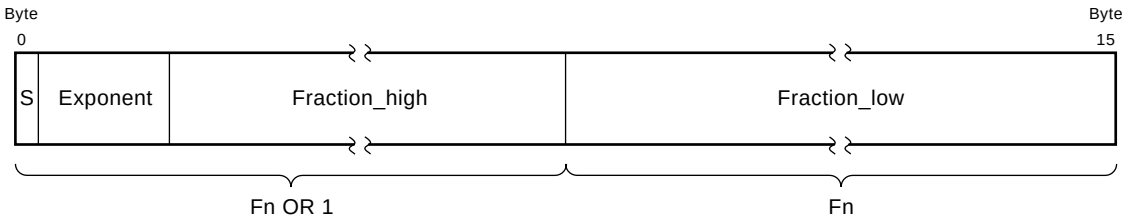


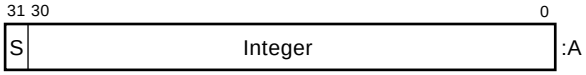
Figure 2–18: X_floating Big-Endian Register Format



2.2.7 Longword Integer Format in Floating-Point Unit

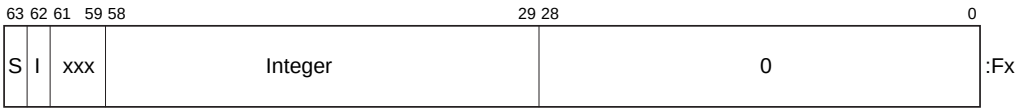
A longword integer operand occupies 32 bits in memory, arranged as shown in Figure 2–19.

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.

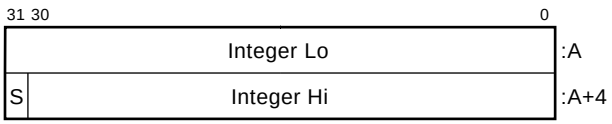
Note:

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

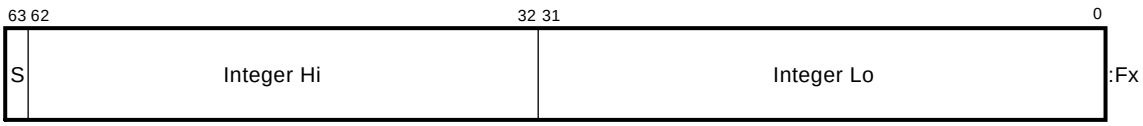
A quadword integer operand occupies 64 bits in memory, arranged as shown in Figure 2–21.

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

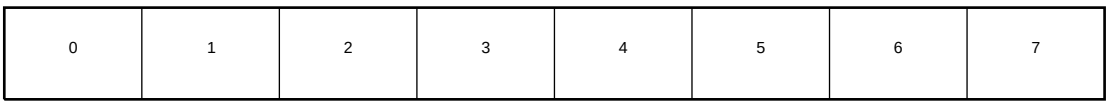
Alpha implementations may include optional big-endian addressing support.

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<2> (bit 2 of the virtual address). This has the effect of accessing the half of a quadword other than the half that would be accessed under the little-endian convention.
- If the big-endian convention is chosen, the word-length load instruction, LDWU, inverts bits va<1:2> (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 instruction, LDBU, inverts bits va<0:2> (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 (EXTxx, INSxx, MSKxx) invert bits Rbv<2:0>. This has the effect of changing a shift of 5 bytes into a shift of 2 bytes, for example.

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

Using either byte numbering convention, it is sometimes necessary to access data that originated on a machine that used the other convention. When this occurs, it is often necessary to swap the bytes within a datum. See Section A.4.3 for a suggested code sequence.

Chapter 3

Instruction Formats

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 Section A.3.5, 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(disp)} to become UNPREDICTABLE.

- LDx_L R31,disp(Rb)

  This instruction produces no useful result since it causes both lock_flag and locked_physical_address to become UNPREDICTABLE.

Unconditional Branch (BR and BSR) and Jump (JMP, JSR, RET, and JSR_COROUTINE) instructions, when R31 is specified as the Ra operand, execute normally and update the PC with the target virtual address. Of course, no PC value can be saved in R31.

3.1.3 Floating-Point Registers

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

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

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

Implementation note:

As described in Section A.3.5, 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 $FFFF_{16}$. The counter wraps no more frequently than 1.5 times the implementation's interval clock interrupt period (which is two thirds of the interval clock interrupt frequency), which guarantees that an interrupt occurs before PCC _CNT overflows twice.

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

Implementation Note:

OpenVMS Alpha and DIGITAL UNIX supply a per-process value in PCC_OFF.

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

The PCC is read by the RPCC instruction. See Section 4.11.8.

3.1.6 Optional Registers

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

3.1.6.1 Memory Prefetch Registers

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

3.1.6.2 VAX Compatibility Register

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

3.2 Notation

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

3.2.1 Operand Notation

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

Table 3–1: Operand Notation

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.

Table 3–5: Operand Access Type Notation (Continued)

Access Type	Meaning
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

Table 3–7: Operators (Continued)

Operator	Meaning
||	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.
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 <n> ↔ x <8n+7:8n>. This correspondence also exists between y and the result. For each bit of y from n = 0 to 7, if y <n> is 0 then byte <n> of x is copied to byte <n> of result, and if y <n> is 1 then byte <n> of result is forced to all zeros.

Table 3–7: Operators (Continued)

Operator	Meaning

CASE	The CASE construct selects one of several actions based on the value of its argument. The form of a case is:
	
	`CASE argument OF`
	`  argvalue1: action_1`
	`  argvalue2: action_2`
	`  ...`
	`  argvaluen:action_n`
	`  [otherwise: default_action]`
	`ENDCASE`
	
	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 ENDCASE (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.
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

Table 3–7: Operators (Continued)

Operator	Meaning
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: `priority_encode( 255 ) = 7`
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 (←) 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.

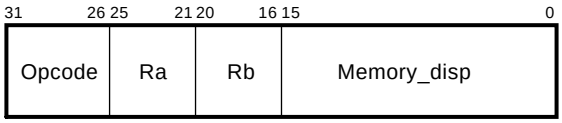
Software Note:

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 ← {Rbv + SEXT(Memory_disp)}
```

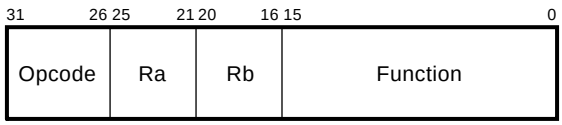
For LDAH the virtual address (va) is computed as follows:

```
va ← {Rbv + SEXT(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.

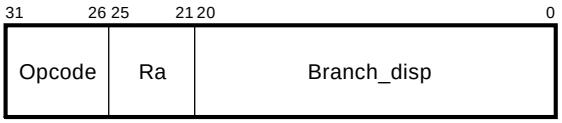
3.3.1.2 Memory Format Jump Instructions

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.

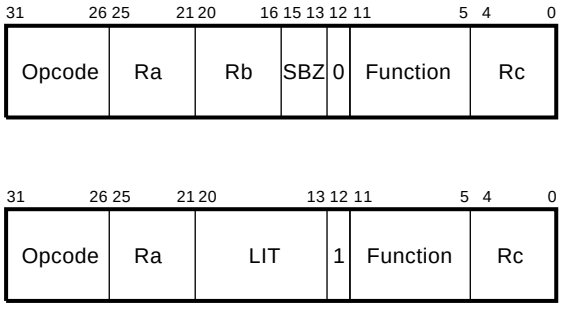
The displacement is treated as a longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits, and added to the updated PC to form the target virtual address. Overflow is ignored in this calculation. The target virtual address (va) is computed as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

```
va ← PC + {4*SEXT(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, 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:

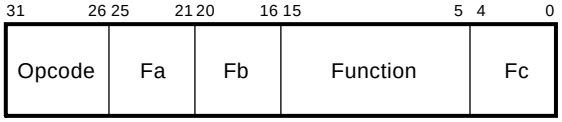
```
IF inst <12> EQ 1 THEN
    Rbv ← ZEXT(inst<20:13>)
ELSE
    IF inst <20:16> EQ 31 THEN
         Rbv ← 0
    ELSE
         Rbv ← Rb
    END
END
```

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, 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.

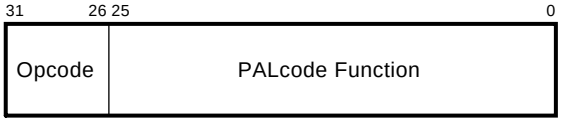
3.3.4.2 Floating-Point/Integer Register Moves

Instructions that move data between a floating-point register file and an integer register file are a subset of 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.

Chapter 4

Instruction Descriptions

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 Alpha operating system. This requirement facilitates the implementation of a floating-point emulator and simplifies context-switching.

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

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

Note:

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

4.1.3 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 layered and application software that executes in User mode may assume that subsetted instructions are emulated by system software. Frequent use of emulation may be significantly slower than using alternative code sequences.

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

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

Exceptions:

None

Instruction mnemonics:

LDA	Load Address
LDAH	Load Address High

Qualifiers:

None

Description:

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

4.2.2 Load Memory Data into Integer Register

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

 CASE
  big_endian_data: va' ← va XOR 000₂     !LDQ
  big_endian_data: va' ← va XOR 100₂     !LDL
  big_endian_data: va' ← va XOR 110₂     !LDWU
  big_endian_data: va' ← va XOR 111₂     !LDBU
  little_endian_data: va' ← va
 ENDCASE

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

Exceptions:

Access Violation

Alignment

Fault on Read

Translation Not Valid

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

In the case of LDQ and LDL, the source operand is fetched from memory, sign-extended, and written to register Ra.

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

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

Notes:

- The word or byte that the LDWU or LDBU instruction fetches from memory is placed in the low (rightmost) word or byte of Ra, with the remaining 6 or 7 bytes set to zero.
- Accesses have byte granularity.
- For big-endian access with LDWU or LDBU, the word/byte remains in the rightmost part of Ra, but the va sent to memory has the indicated bits inverted. See Operation section, above.
- No sparse address space mechanisms are allowed with the LDWU and LDBU instructions.

Implementation Notes:

- The LDWU and LDBU instructions are supported in hardware on Alpha implementations for which the AMASK instruction returns bit 0 set. LDWU and LDBU are supported with software emulation in Alpha implementations for which AMASK does not return bit 0 set. 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 ← {{Rbv + SEXT(disp)} AND NOT 7}
Ra ← (va)<63:0>
```

Exceptions:

Access Violation

Fault on Read

Translation Not Valid

Instruction mnemonics:

LDQ_U	Load Unaligned Quadword from Memory to Register

Qualifiers:

None

Description:

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

4.2.4 Load Memory Data into Integer Register Locked

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

CASE
  big_endian_data:  va' ← va XOR 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<2> (bit 2 of the virtual address) is inverted, and any memory management fault is reported for va (not va'). The source operand is fetched from memory, sign-extended for LDL_L, and written to register Ra.

When a LDx_L instruction is executed without faulting, the processor records the target physical address in a per-processor locked_physical_address register and sets the per-processor lock_flag.

If the per-processor lock_flag is (still) set when a STx_C instruction is executed (accessing within the same 16-byte naturally aligned block as the LDx_L), the store occurs; otherwise, it does not occur, as described for the STx_C instructions. The behavior of 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_conditional, or executing a WH64 instruction that modifies data on processor *B*. A processor's locked range is the aligned block of 2**N bytes that includes the locked_physical_address. The 2**N value is implementation dependent. It is at least 16 (minimum lock range is an aligned 16-byte block) and is at most the page size for that implementation (maximum lock range is one physical page).

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

The sequence:

```
LDx_L
Modify
STx_C
BEQ xxx
```

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

Notes:

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

  Executing a LDx_L instruction on one processor does not affect any architecturally visible state on another processor, and in particular cannot cause 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, STQ_U, WH64) is executed on the given processor between the LDx_L and the STx_C, the sequence above may always fail on some implementations; hence, no useful program should do this.
- If a branch is taken between the LDx_L and the STx_C, the sequence above may always fail on some implementations; hence, no useful program should do this. (CMOVxx may be used to avoid branching.)
- If a subsetted instruction (for example, floating-point) is executed between the LDx_L and the STx_C, the sequence above may always fail on some implementations because of the Illegal Instruction Trap; hence, no useful program should do this.
- If an instruction with an unused function code is executed between the LDx_L and the STx_C, the sequence above may always fail on some implementations because an instruction with an unused function code is UNPREDICTABLE.
- If a large number of instructions are executed between the LDx_L and the STx_C, the sequence above may always fail on some implementations because of a timer interrupt always clearing the lock_flag before the sequence completes; hence, no useful program should do this.
- Hardware implementations are encouraged to lock no more than 128 bytes. Software implementations are encouraged to separate locked locations by at least 128 bytes from other locations that could potentially be written by another processor while the first location is locked.
- Execution of a WH64 instruction on processor *A* to a region within the lock range of processor *B*, where the execution of the WH64 changes the contents of memory, causes the lock_flag on processor *B* to be cleared. If the WH64 does not change the contents of memory on processor *B*, it need not clear the lock_flag.

Implementation Notes:

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

All implementations should guarantee that at least 40 non-subsetted operate instructions can be executed between timer interrupts.

4.2.5 Store Integer Register Data into Memory Conditional

Format:

```
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<2> (bit 2 of the virtual address) is inverted, and any memory management fault is reported for va (not va').

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

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

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

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

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

Notes:

- Software will not emulate unaligned STx_C instructions.
- Each implementation must do the test and store atomically, as illustrated in the following two examples. (See Section 5.6.1 for complete information.)
  - If two processors attempt STx_C instructions to the same lock range and that lock range was accessed by both processors' preceding LDx_L instructions, exactly one of the stores succeeds.
  - A processor executes a LDx_L/STx_C sequence and includes an MB between the LDx_L to a particular address and the *successful* STx_C to a different address (one that meets the constraints required for predictable behavior). That instruction sequence establishes an access order under which a store operation by another processor to that lock range occurs before the LDx_L or after the STx_C.
- If the virtual and physical addresses for a LDx_L and STx_C sequence are not within the same naturally aligned 16-byte sections of virtual and physical memory, that sequence may always fail, or may succeed despite another processor's store to the lock range; hence, no useful program should do this.
- The following sequence should not be used:

```
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 ← {Rbv + SEXT(disp)}

CASE
 big_endian_data: va' ← va XOR 000₂      !STQ
 big_endian_data: va' ← va XOR 100₂      !STL
 big_endian_data: va' ← va XOR 110₂      !STW
 big_endian_data: va' ← va XOR 111₂      !STB
 little_endian_data: va' ← va
ENDCASE

 (va') ← Rav                             !STQ
 (va')<31:00> ← Rav<31:0>                !STL
 (va')<15:00> ← Rav<15:0>                !STW
 (va')<07:00> ← Rav<07:0>                !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 returns bit 0 set. STB and STW are supported with software emulation in Alpha implementations for which AMASK does not return bit 0 set. 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 ← {{Rbv + SEXT(disp)} AND NOT 7}
(va)<63:0> ← Rav<63:0>
```

Exceptions:

Access Violation

Fault on Write

Translation Not Valid

Instruction mnemonics:

STQ_U	Store Unaligned Quadword from Register to Memory

Qualifiers:

None

Description:

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

4.3 Control Instructions

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

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

- 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

Table 4–3: Control Instructions Summary (Continued)

Mnemonic	Operation
BNE	Branch if Register Not Equal to Zero
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.2.2 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.

Count instruction (CIX) extension implementation note:

The CIX extension to the architecture provides the CTLZ, CTPOP, and CTTZ instructions. Alpha processors for which the AMASK instruction returns bit 2 set implement these instructions. Those processors for which AMASK does not return bit 2 set can take an Illegal Instruction trap, and software can emulate their function, if required. AMASK is described in Sections 4.11.1 and D.3.

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:

```
Rc ← SEXT( (Rav + Rbv)<31:0>)
```

Exceptions:

Integer Overflow

Instruction mnemonics:

ADDL	Add Longword

Qualifiers:

Integer Overflow Enable (/V)

Description:

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

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

4.4.2 Scaled Longword Add

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

4.4.3 Quadword Add

Format:

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

Operation:

```
Rc ← Rav + Rbv
```

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:

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

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.

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.

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 ← 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 ← Rav * Rbv
```

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

The NOT function can be performed by doing an ORNOT with zero (Ra = R31).

4.5.2 Conditional Move Integer

Format:

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

Operation:

```
IF TEST(Rav, Condition_based_on_Opcode) THEN

  Rc ← Rbv
```

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

Register Ra is tested. If the specified relationship is true, the value Rbv is written to register Rc.

Notes:

Except that it is likely in many implementations to be substantially faster, the instruction:

```
CMOVEQ Ra,Rb,Rc
```

is exactly equivalent to:

```
      BNE Ra,label
      OR  Rb,Rb,Rc
label: ...
```

For example, a branchless sequence for:

```
R1=MAX(R1,R2)
```

is:

```
CMPLT  R1,R2,R3                    ! R3=1 if R1<R2
CMOVNE R3,R2,R1                    ! Move R2 to R1 if R1<R2
```

4.5.3 Shift Logical

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

SLL	Shift Left Logical
SRL	Shift Right Logical

Qualifiers:

None

Description:

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

4.5.4 Shift Arithmetic

Format:

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

Operation:

```
Rc ← 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 instruction reports whether a particular Alpha implementation supports the BWX extension. AMASK is described in Sections 4.11.1 and D.3.

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 Section A.4.1.

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
INSLL	Insert Longword Low
INSQL	Insert Quadword Low

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

Mnemonic	Operation
INSWH	Insert Word High
INSLH	Insert Longword High
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 111(2)
 little_endian_data: Rbv' ← Rbv
ENDCASE

CASE
  EXTBL: byte_mask ← 0000 0001(2)
  EXTWx: byte_mask ← 0000 0011(2)
  EXTLx: byte_mask ← 0000 1111(2)
  EXTQx: byte_mask ← 1111 1111(2)
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
EXTWH	Extract Word High
EXTLH	Extract Longword High
EXTQH	Extract Quadword High

Qualifiers:

None

Description:

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

Notes:

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

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

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

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

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

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

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

```
LDQ_U  R1,  X(R11)        ; Ignores va<2:0>, R1 = CBAx xxxx
LDQ_U  R2,  X+3(R11)      ; Ignores va<2:0>, R2 = yyyy yyyD
LDA    R3,  X(R11)        ; R3<2:0> = (X mod 8) = 5
EXTLL  R1,  R3, R1        ; R1 = 0000 0CBA
EXTLH  R2,  R3, R2        ; R2 = 0000 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 long-word-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 = xxxxxABC
LDQ_U  R2, X+7(R11)       ; Ignores va<2:0>, R2 = DEFGHyyy
LDA    R3, X+7(R11)       ; R3<2:0> = 4, shift will be 3 bytes
EXTQH  R1, R3, R1         ; R1 = ABC0 0000
EXTQL  R2, R3, R2         ; R2 = 000D EFGH
OR     R1, R2, R1         ; R1 = ABCD EFGH
```

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

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 111(2)
 little_endian_data: Rbv' ← Rbv
ENDCASE

CASE
  INSBL: byte_mask ← 0000 0000 0000 0001(2)
  INSWx: byte_mask ← 0000 0000 0000 0011(2)
  INSLx: byte_mask ← 0000 0000 0000 1111(2)
  INSQx: byte_mask ← 0000 0000 1111 1111(2)
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 111(2)
 little_endian_data: Rbv'← Rbv
ENDCASE

CASE
  MSKBL: byte_mask ← 0000 0000 0000 0001(2)
  MSKWx: byte_mask ← 0000 0000 0000 0011(2)
  MSKLx: byte_mask ← 0000 0000 0000 1111(2)
  MSKQx: byte_mask ← 0000 0000 1111 1111(2)
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 mod 8) = 5, the value of the aligned quadword containing X(R11) is CBAx xxxx, the value of the aligned quadword containing X+7(R11) is yyyH GFED, the value to be stored from R5 is HGFE DCBA, and the datum is little-endian. Slight modifications similar to those in Section 4.6.2 apply to big-endian data.

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

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

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

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

```
LDA    R6, X(R11)         ; R6<2:0> = (X mod 8) = 5
LDQ_U  R2, X+3(R11)       ; Ignores va<2:0>, R2 = yyyy yyyD
LDQ_U  R1, X(R11)         ; Ignores va<2:0>, R1 = CBAx xxxx
INSLH  R5, R6, R4         ; R4 = 0000 000D
INSLL  R5, R6, R3         ; R3 = CBA0 0000
MSKLH  R2, R6, R2         ; R2 = yyyy yyy0
MSKLL  R1, R6, R1         ; R1 = 000x xxxx
OR     R2, R4, R2         ; R2 = yyyy yyyD
OR     R1, R3, R1         ; R1 = CBAx xxxx
STQ_U  R2, X+3(R11)       ; Must store high then low for
STQ_U  R1, X(R11)         ; degenerate case of aligned
```

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

```
LDA    R6, X(R11)          ; R6<2:0> = (X mod 8) = 5
LDQ_U  R2, X+1(R11)        ; Ignores va<2:0>, R2 = yBAx xxxx
LDQ_U  R1, X(R11)          ; Ignores va<2:0>, R1 = yBAx xxxx
INSWH  R5, R6, R4          ; R4 = 0000 0000
INSWL  R5, R6, R3          ; R3 = 0BA0 0000
MSKWH  R2, R6, R2          ; R2 = yBAx xxxx
MSKWL  R1, R6, R1          ; R1 = y00x xxxx
OR     R2, R4, R2          ; R2 = yBAx xxxx
OR     R1, R3, R1          ; R1 = yBAx xxxx
STQ_U  R2, X+1(R11)        ; Must store high then low for
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 returns bit 0 set. SEXTB and SEXTW are supported with software emulation in Alpha implementations for which AMASK does not return bit 0 set. 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. Single-precision operations applied to non-canonical 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>.

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, INVD, 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 infinites, NaNs, or denormals, but do include minus zero.

denormal

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

dirty zero

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

infinity

An IEEE floating-point bit pattern that represents plus or minus infinity.

LSB

The least significant bit. For a positive finite representable number A, A + 1 LSB is the next larger representative number, and A + ½ 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 – ½ 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 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) (0.8988465674311578540 7e308)
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‡)

† (1.189731495357231765085759326628007 02e4932)

‡ (3.362103143112093506262677817321752 60e–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 ≥ MAX + 1/2 LSB in magnitude to an overflow; maps true results < MIN – 1/4 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 ≥ MAX + 1 LSB in magnitude to an overflow; maps true results < MIN in magnitude to an underflow.

IEEE Rounding Modes

For IEEE floating-point operations, four rounding modes are provided: normal rounding (unbiased round to nearest), rounding toward minus infinity, round toward zero, and rounding toward plus infinity. The first three can be specified in the 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 ≥ MAX + 1/2 LSB in magnitude to an overflow; maps true results < MIN – 1/2 LSB in magnitude to an underflow.

Plus infinity IEEE rounding maps the true result to the larger of two surrounding representable results; maps true results > MAX in magnitude to an overflow; maps positive true results ≤ +MIN – 1 LSB to an underflow; and maps negative true results > –MIN to an underflow.

Minus infinity IEEE rounding maps the true result to the smaller of two surrounding representable results; maps true results > MAX in magnitude to an overflow; maps positive true results < +MIN to an underflow; and maps negative true results ≥ –MIN + 1 LSB to an underflow.

Chopped IEEE rounding maps the true result to the smaller in magnitude of two surrounding representable results; maps true results ≥ MAX + 1 LSB in magnitude to an overflow; and maps non-zero true results < 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 /S	Imprecise Precise exception completion
Underflow enabled	/U /SU	Imprecise Precise exception completion
Integer overflow disabled	No qualifier /S	Imprecise Precise exception completion
Integer overflow enabled	/V /SV	Imprecise 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 Section B.2.1, 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 /SU	Imprecise Precise exception completion
Underflow enabled and inexact enabled	/SUI	Precise exception completion
Integer overflow disabled and inexact disabled	No qualifier	Imprecise

Table 4–9: Summary of IEEE Trapping Modes (Continued)

Trap Mode	Notation	Meaning
Integer overflow enabled and inexact disabled	/V /SV	Imprecise 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 returns with bit 9 clear, floating-point traps are imprecise. Exception completion requires that generated code must obey the trap shadow rules in Section 4.7.7.3.1, with a trap shadow length as described in Section 4.7.7.3.2.
- If AMASK returns with bit 9 set, 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 4.7.7.3.1 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 4–10 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 an 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)	• 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	• 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.

Table 4–10: Trap Shadow Length Rules (Continued)

Floating-Point Instruction Group	Trap Shadow Extends Until Any of the Following Occurs:
Floating-point SQRTx	• 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 *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors Hardware Reference Manual, EC-QD2RA-TE*.

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

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

If an underflow occurs, a true zero (64 bits of zero) is always stored in the result register. 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 4.7.7.4 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, INVD, 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 UNDZ 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 UNDZ (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 UNDZ 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 <12:11> = 11), the rounding mode to be used for the instruction is derived from the FPCR register. The layout of the rounding mode bits and their assignments matches exactly the format used in the 11-bit function field of the floating-point operate instructions. The function field is described in Section 4.7.9.

In addition, the FPCR gives a summary of each exception type for the exception conditions detected by all IEEE floating-point operates thus far, as well as an overall summary bit that indicates whether any of these exception conditions has been detected. The individual exception bits match exactly in purpose and order the 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

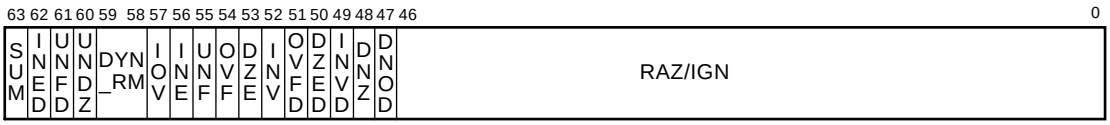


Table 4–11: Floating-Point Control Register (FPCR) Bit Descriptions

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>.
62	Inexact Disable (INED)†. Suppress INE trap and place correct IEEE nontrapping result in the destination register.
61	Underflow Disable (UNFD)†. Suppress UNF trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping result. The correct result value is determined according to the value of the UNDZ bit.
60	Underflow to Zero (UNDZ)†. When set together with UNFD, on underflow, the hardware places a true zero (64 bits of zero) in the destination register rather than the result specified by 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:

DYN	IEEE Rounding Mode Selected
00	Chopped rounding mode
01	Minus infinity
10	Normal rounding
11	Plus infinity

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

Bit	Description (Meaning When Set)
57	Integer Overflow (IOV). An integer arithmetic operation or a conversion from floating to integer 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.
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 (INVD)†. 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 Alpha 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 Section B.2.

   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

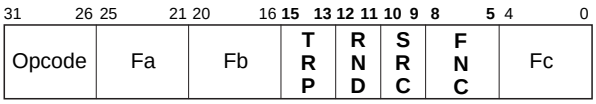


Table 4–12: IEEE Floating-Point Function Field Bit Summary

Bits	Field	Meaning†
15–13	TRP	Trapping modes:

Contents	Meaning for Opcodes 14_{16} and 16_{16}
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

12–11 RND Rounding modes:

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:

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

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

Bits	Field	Meaning†		
8–5	FNC	Instruction class:		
		Contents	**Meaning for Opcode 16_{16}**	**Meaning for Opcode 14_{16}**
		0000	ADDx	Reserved
		0001	SUBx	Reserved
		0010	MULx	Reserved
		0011	DIVx	Reserved
		0100	CMPxUN	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 Section C.1.

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

Bits	Field	Meaning
15–13	TRP	Trapping modes:

Contents	Meaning for Opcodes 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

12–11	RND	Rounding modes:

Contents	Meaning for Opcodes 15_{16} and 14_{16}
00	Chopped (/C)
01	UNPREDICTABLE
10	Normal (default)
11	UNPREDICTABLE

10–9	SRC	Source datatype:†

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

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

Bits	Field	Meaning		
8–5	FNC	Instruction class:		
		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	SQRTF/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 14_{16}.

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 Section B.1.

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 ← {Rbv + SEXT(disp)}

CASE
 big_endian_data:  va' ← va XOR 100₂
 little_endian_data: va' ← va
ENDCASE

Fa ← (va')<15> || MAP_F((va')<14:7>) || (va')<6:0> ||
      (va')<31:16> || 0<28:0>
```

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDF Load F_floating

Qualifiers:

None

Description:

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

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

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

4.8.2 Load G_floating

Format:

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

Operation:

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

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDG	Load G_floating (Load D_floating)

Qualifiers:

None

Description:

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

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

4.8.3 Load S_floating

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

CASE
 big_endian_data:  va' ← va XOR 100(2)
 little_endian_data: va' ← va
ENDCASE

Fa ← (va')<31> || MAP_S((va')<30:23>) || (va')<22:0> || 0<28:0>
```

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDS	Load S_floating (Load Longword Integer)

Qualifiers:

None

Description:

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

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

4.8.4 Load T_floating

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

Fa ← (va)<63:0>
```

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDT	Load T_floating (Load Quadword Integer)

Qualifiers:

None

Description:

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

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

4.8.5 Store F_floating

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

CASE
 big_endian_data:  va' ← va XOR 100(2)
 little_endian_data: va' ← va
ENDCASE

(va')<31:0> ← Fav<44:29> || Fav<63:62> || Fav<58:45>
```

Exceptions:

Access Violation

Fault on Write

Alignment

Translation Not Valid

Instruction mnemonics:

STF	Store F_floating

Qualifiers:

None

Description:

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

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

4.8.6 Store G_floating

Format:

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

Operation:

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

Exceptions:

Access Violation
Fault on Write
Alignment
Translation Not Valid

Instruction mnemonics:

STG	Store G_floating (Store D_floating)

Qualifiers:

None

Description:

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

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

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

Exceptions:

Access Violation
Fault on Write
Alignment
Translation Not Valid

Instruction mnemonics:

STT Store T_floating (Store Quadword Integer)

Qualifiers:

None

Description:

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

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

4.9 Branch Format Floating-Point Instructions

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

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

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

The floating-point branch operations are summarized in Table 4–15:

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 +/–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, squre 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.

Floating-point convert and square-root (FIX) extension implementation note:

The FIX extension to the architecture provides the FTOIx, ITOFx, and SQRTx instructions. Alpha processors for which the AMASK instruction returns bit 1 set implement these instructions. Those processors for which AMASK does not return bit 1 set can take an Illegal Instruction trap, and software can emulate their function, if required. AMASK is described in Sections 4.11.1 and D.3.

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

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

Mnemonic	Operation	Subset
Arithmetic Operations		
ADDF	Add F_floating	VAX
ADDG	Add G_floating	VAX
ADDS	Add S_floating	IEEE
ADDT	Add T_floating	IEEE
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

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

Mnemonic	Operation	Subset
Arithmetic Operations		
MULF	Multiply F_floating	VAX
MULG	Multiply G_floating	VAX
MULS	Multiply S_floating	IEEE
MULT	Multiply T_floating	IEEE
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:

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

Operation:

```
IF TEST(Fav, Condition_based_on_Opcode) THEN

  Fc ← Fbv
```

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

Register Fa is tested. If the specified relationship is true, register Fb is written to register Fc; otherwise, the move is suppressed and register Fc is unchanged. The test is based on the sign bit and whether the rest of the register is all zero bits, as described for floating branches in Section 4.9.

Notes:

Except that it is likely in many implementations to be substantially faster, the instruction:

```
FCMOVxx Fa,Fb,Fc
```

is exactly equivalent to:

```
        FByy Fa,label          ! yy = NOT xx
        CPYS Fb,Fb,Fc
label:  ...
```

For example, a branchless sequence for:

```
F1=MAX(F1,F2)
```

is:

```
CMPxLT  F1,F2,F3       ! F3=one if F1<F2; x=F/G/S/T
FCMOVNE F3,F2,F1       ! Move F2 to F1 if F1<F2
```

4.10.4 Move from/to Floating-Point Control Register

Format:

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:

Fc ← Fav + Fbv

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:

Fc ← Fav + Fbv

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

ADDS	Add S_floating
ADDT	Add T_floating

Qualifiers:

Rounding:	Dynamic (/D)
	Minus infinity (/M)
	Chopped (/C)
Trapping:	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 0000(16)
ELSE
  Fc ← 0000 0000 0000 0000(16)
```

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 must be provided by an operating system (OS) completion handler, since NaNs trap.) Comparisons ignore the sign of zero, so +0 = –0.

Comparisons with plus and minus infinity execute normally and do not take an invalid operation trap.

Notes:

- In order to use CMPTxx with 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:

```
Fc ← {conversion of Fbv}
```

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:

```
Fc ← {conversion of Fbv<63:0>}
```

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:

```
Fc ← {conversion of Fbv}
```

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 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:

```
Fc ← {conversion of Fbv}
```

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:

```
Fc ← {conversion of Fbv<63:0>}
```

Exceptions:

Inexact Result

Instruction mnemonics:

CVTQS	Convert Quadword to S_floating
CVTQT	Convert Quadword to T_floating

Qualifiers:

Rounding:	Dynamic (/D)
	Minus infinity (/M)
	Chopped (/C)
Trapping:	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:

```
Fc ← {conversion of Fbv}
```

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:

```
Fc ← {conversion of Fbv}
```

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:

```
Fc ← Fav / Fbv
```

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 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:

Fc ← Fav / Fbv

Exceptions:

Invalid Operation
Division by Zero
Overflow
Underflow
Inexact Result

Instruction mnemonics:

DIVS	Divide S_floating
DIVT	Divide T_floating

Qualifiers:

Rounding:	Dynamic (/D)
	Minus infinity (/M)
	Chopped (/C)
Trapping:	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.

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.

4.10.20 VAX Floating Multiply

Format:

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

Operation:

Fc ← Fav * Fbv

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 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:

Fc ← Fav * Fbv

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

MULS	Multiply S_floating
MULT	Multiply T_floating

Qualifiers:

Rounding:	Dynamic (/D)
	Minus infinity (/M)
	Chopped (/C)
Trapping:	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:

```
Fc ← Fb ** (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.

4.10.23 IEEE Floating Square Root

Format:

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

Operation:

```
Fc ← Fb ** (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.

4.10.24 VAX Floating Subtract

Format:

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

Operation:

Fc ← Fav - Fbv

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 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:

Fc ← Fav - Fbv

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
RPCC	Read Processor Cycle Counter
TRAPB	Trap Barrier
WH64	Write Hint — 64 Bytes
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 ← Rbv AND {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 Section D.3.

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), and 21264 (EV6), AMASK correctly indicates support for architecture extensions by copying Rbv to Rc and clearing appropriate bits.

Bits are assigned and placed in Appendix D for architecture extensions as ECOs for those extensions are passed. The low 8 bits are reserved for standard architecture extensions so they can be tested with a literal; application-specific extensions are assigned from bit 8 upward.

4.11.2 Call Privileged Architecture Library

Format:

CALL_PAL	fnc.ir	!PAL format

Operation:

```
{Stall instruction issuing until all
prior instructions are guaranteed to
complete without incurring exceptions.}
{Trap to PALcode.}
```

Exceptions:

None

Instruction mnemonics:

CALL_PAL	Call Privileged Architecture Library

Qualifiers:

None

Description:

The CALL_PAL instruction is not issued until all previous instructions are guaranteed to complete without exceptions. If an exception occurs, the continuation PC in the exception stack frame points to the CALL_PAL instruction. The CALL_PAL instruction causes a trap to PALcode.

4.11.3 Evict Data Cache Block

Format:

```
ECB          (Rb.ab)          ! Memory format
```

Operation:

```
va ← Rbv

IF { va maps to memory space } THEN
 Prepare to reuse cache resources that are occupied by the
 the addressed byte.
END
```

Exceptions:

None

Instruction mnemonics:

ECB	Evict Cache Block

Qualifiers:

None

Description:

The ECB instruction provides a hint that the addressed location will not be referenced again in the near future, so any cache space it occupies should be made available to cache other memory locations. If the cache copy of the location is dirty, the processor may start writing it back; if the cache has multiple sets, the processor may arrange for the set containing the addressed byte to be the next set allocated.

The ECB instruction does not generate exceptions; if it encounters data address translation errors (access violation, translation not valid, and so forth) during execution, it is treated as a NOP.

If the address maps to non-memory-like (I/O) space, ECB is treated as a NOP.

Software Note:

- ECB makes a particular cache location available for reuse by evicting and invalidating its contents. The intent is to give software more control over cache allocation policy in set-associative caches so that "useful" blocks can be retained in the cache.
- ECB is a performance hint — it does not serialize the eviction of the addressed cache block with any preceding or following memory operation.

- ECB is not intended for flushing caches prior to power failure or low power operation — CFLUSH is intended for that purpose.

Implementation Note:

Implementations with set-associative caches are encouraged to update their allocation pointer so that the next D-stream reference that misses the cache and maps to this line is allocated into the vacated set.

4.11.4 Exception Barrier

Format:

```
EXCB                                                    ! Memory format
```

Operation:

```
{EXCB does not appear to issue until completion of all
 exceptions and dependencies on the Floating-point Control
 Register (FPCR) from prior instructions.}
```

Exceptions:

None

Instruction mnemonics:

EXCB	Exception Barrier

Qualifiers:

None

Description:

The EXCB instruction allows software to guarantee that in a pipelined implementation, all previous instructions have completed any behavior related to exceptions or rounding modes before any instructions after the EXCB are issued.

In particular, all changes to the Floating-point Control Register (FPCR) are guaranteed to have been made, whether or not there is an associated exception. Also, all potential floating-point exceptions and integer overflow exceptions are guaranteed to have been taken. EXCB is thus a superset of TRAPB.

If a floating-point exception occurs for which trapping is enabled, the EXCB instruction acts like a fault. In this case, the value of the Program Counter reported to the program may be the address of the EXCB instruction (or earlier) but is never the address of an instruction following the EXCB.

The relationship between EXCB and the FPCR is described in Section 4.7.8.1.

4.11.5 Prefetch Data

Format:

FETCHx	0(Rb.ab)	!Memory format

Operation:

```
va ← {Rbv}
{Optionally prefetch aligned 512-byte block surrounding va.}
```

Exceptions:

None

Instruction mnemonics:

FETCH	Prefetch Data
FETCH_M	Prefetch Data, Modify Intent

Qualifiers:

None

Description:

The virtual address is given by Rbv. This address is used to designate an aligned 512-byte block of data. An implementation may optionally attempt to move all or part of this block (or a larger surrounding block) of data to a 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.3.5) should be used to overlap such shorter latencies.

Existing Alpha implementations (through the 21264) 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 Chapter 5 for details on using the MB instruction to serialize these accesses. Chapter 5 also details coordinating memory accesses across processors.

Note that MB ensures serialization only; it does not necessarily accelerate the progress of memory operations.

4.11.8 Read Processor Cycle Counter

Format:

RPCC	Ra.wq	!Memory format

Operation:

```
Ra ← {cycle counter}
```

Exceptions:

None

Instruction mnemonics:

RPCC	Read Processor Cycle Counter

Qualifiers:

None

Description:

Register Ra is written with the processor cycle counter (PCC). The PCC register consists of two 32-bit fields. The low-order 32 bits (PCC<31:0>) are an unsigned, wrapping counter, PCC_CNT. The high-order 32 bits (PCC<63:32>), PCC_OFF, are operating-system dependent in their implementation.

See Section 3.1.5 for a description of the PCC.

If an operating system uses PCC_OFF to calculate the per-process or per-thread cycle count, that count must be derived from the 32-bit sum of PCC_OFF and PCC_CNT. The following example computes that cycle count, modulo 2**32, and returns the count value in R0. Notice the care taken not to cause an unwanted sign extension.

```
RPCC  R0                ; Read the process cycle counter
SLL   R0, #32, R1       ; Line up the offset and count fields
ADDQ  R0, R1, R0        ; Do add
SRL   R0, #32, R0       ; Zero extend the count to 64  bits
```

The following example code returns the value of PCC_CNT in R0<31:0> and all zeros in R0<63:32>.

```
RPCC  R0
ZAPNOT R0,#15,R0
```

4.11.9 Trap Barrier

Format:

```
TRAPB                                                    !Memory format
```

Operation:

```
{TRAPB does not appear to issue until all prior instructions
 are guaranteed to complete without causing any arithmetic traps}.
```

Exceptions:

None

Instruction mnemonics:

TRAPB	Trap Barrier

Qualifiers:

None

Description:

The TRAPB instruction allows software to guarantee that in a pipelined implementation, all previous arithmetic instructions will complete without incurring any arithmetic traps before the TRAPB or any instructions after it are issued.

If an arithmetic exception occurs for which trapping is enabled, the TRAPB instruction acts like a fault. In this case, the value of the Program Counter reported to the program may be the address of the TRAPB instruction (or earlier) but is never the address of the instruction following the TRAPB.

This fault behavior by TRAPB allows software, using one TRAPB instruction for each exception domain, to isolate the address range in which an exception occurs. If the address of the instruction following the TRAPB were allowed, there would be no way to distinguish an exception in the address range preceding a label from an exception in the range that includes the label along with the faulting instruction and a branch back to the label. This case arises when the code is not following 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.

4.11.10 Write Hint

Format:

WH64	(Rb.ab)	! Memory format

Operation:

```
va ← Rbv
IF { va maps to memory space } THEN
 Write UNPREDICTABLE data to the aligned 64-byte region
 containing the addressed byte.
END
```

Exceptions:

None

Instruction mnemonics:

WH64	Write Hint - 64 Bytes

Qualifiers:

None

Description:

The WH64 instruction provides a hint that the current contents of the aligned 64-byte block containing the addressed byte will never be read again but will be overwritten in the near future.

The processor may allocate cache resources to hold the block without reading its previous contents from memory; the contents of the block may be set to any value that does not introduce a security hole, as described in Section 1.6.3.

The WH64 instruction does not generate exceptions; if it encounters data address translation errors (access violation, translation not valid, and so forth), it is treated as a NOP.

If the address maps to non-memory-like (I/O) space, WH64 is treated as a NOP.

Software Note:

This instruction is a performance hint that should be used when writing a large continuous region of memory. The intended code sequence consists of one WH64 instruction followed by eight quadword stores for each aligned 64-byte region to be written.

Sometimes, the UNPREDICTABLE data will exactly match some or all of the previous contents of the addressed block of memory.

Implementation Note:

If the 64-byte region containing the addressed byte is not in the data cache, implementations are encouraged to allocate the region in the data cache without first reading it from memory. However, if any of the addressed bytes exist in the caches of other processors, they must be kept coherent with respect to those processors.

Processors with cache blocks smaller than 64 bytes are encouraged to implement WH64 as defined. However, they may instead implement the instruction by allocating a smaller aligned cache block for write access or by treating WH64 as a NOP.

Processors with cache blocks larger than 64 bytes are also encouraged to implement WH64 as defined. However, they may instead treat WH64 as a NOP.

4.11.11 Write Memory Barrier

Format:

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 5.5.4, 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 not a permanent part of the architecture and will not be available in some future implementations. They are intended to preserve customer assumptions about VAX instruction atomicity in porting code from VAX to Alpha.

These instructions should be generated only by the VAX-to-Alpha software translator; they should never be used in native Alpha code. Any native code that uses them may cease to work.

Table 4–18: VAX Compatibility Instructions Summary

Mnemonic	Operation
RC	Read and Clear
RS	Read and Set

4.12.1 VAX Compatibility Instructions

Format:

```
Rx          Ra.wq                    !Memory format
```

Operation:

```
Ra ← intr_flag
intr_flag ← 0                        !RC
intr_flag ← 1                        !RS
```

Exceptions:

None

Instruction mnemonics:

RC	Read and Clear
RS	Read and Set

Qualifiers:

None

Description:

The intr_flag is returned in Ra and then cleared to zero (RC) or set to one (RS).

These instructions may be used to determine whether the sequence of Alpha instructions between RS and RC (corresponding to a single VAX instruction) was executed without interruption or exception.

Intr_flag is a per-processor state bit. The intr_flag is cleared if that processor encounters a CALL_PAL REI instruction.

It is UNPREDICTABLE whether a processor's intr_flag is affected when that processor executes an LDx_L or STx_C instruction. A processor's intr_flag is not affected when that processor executes a normal load or store instruction.

A processor's intr_flag is not affected when that processor executes a taken branch.

Notes:

- These instructions are intended *only* for use by the VAX-to-Alpha software translator; they should never be used by native code.

4.13 Multimedia (Graphics and Video) Support

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 maximium 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 returns bit 8 set implement these instructions. Those processors for which AMASK does not return bit 8 set 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.rq,Rc.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.

Chapter 5

System Architecture and Programming Implications

5.1 Introduction

Portions of the Alpha architecture have implications for programming, and the system structure, of both uniprocessor and multiprocessor implementations. Architectural implications considered in the following sections are:

- Physical address space behavior
- Caches and write buffers
- Translation buffers and virtual caches
- Data sharing
- Read/write ordering
- Arithmetic traps

To meet the requirements of the Alpha architecture, software and hardware implementors need to take these issues into consideration.

5.2 Physical Address Space Characteristics

Alpha physical address space is divided into four equal-size regions. The regions are delineated by the two most significant, implemented, physical address bits. Each region's characteristics are distinguished by the coherency, granularity, and width of memory accesses, and whether the region exhibits memory-like behavior or non-memory-like behavior.

5.2.1 Coherency of Memory Access

Alpha implementations must provide a coherent view of memory, in which each write by a processor or I/O device (hereafter, called "processor") becomes visible to all other processors. No distinction is made between coherency of "memory space" and "I/O space."

Memory coherency may be provided in different ways for each of the four physical address regions.

Possible per-region policies include, but are not restricted to:

- 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 LDx_L/STx_C sequence.

5.5.2 Atomic Update of a Single Datum

The load-locked/store-conditional instructions may be used to perform an atomic update of a shared aligned longword or quadword. ("Update" means that the new value is a function of the old value.)

The following sequence performs a read-modify-write operation on location *x*. Only register-to-register operate instructions and branch fall-throughs may occur in the sequence:

```
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 LDx_L and STx_C will never complete, because periodic timer interrupts will always occur before the sequence completes. The rules in Section A.5 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.

- Both conditional branches are forward branches, so they are properly predicted not to be taken (to match the common case of no contention for the lock).
- 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 5.5.4. 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 ($1 \rightarrow 1$ does not change state).

5.5.4 Ordering Considerations for Shared Data Structures

A critical section sequence, such as shown in Section 5.5.3, is conceptually only three steps:

1. Acquire software lock
2. Critical section — read/write shared data
3. Clear software lock

In the absence of explicit instructions to the contrary, the Alpha architecture allows reads and writes to be reordered. While this may allow more implementation speed and overlap, it can also create undesired side effects on shared data structures. Normally, the critical section just described would have two instructions added to it:

```
<acquire software lock>
MB (memory barrier #1)
<critical section – read/write shared data>
MB (memory barrier #2)
<clear software lock>
<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 $x \le z < x+m$. 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)	⇐ if overlap		⇐ if overlap	⇐	⇐
Pi:R<m>(x,a)		⇐ if overlap	⇐ if overlap	⇐	⇐
Pi:W<m>(x,a)			⇐ if overlap	⇐	⇐
Pi:MB		⇐	⇐	⇐	⇐
Pi:IMB	⇐	⇐	⇐	⇐	⇐

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 (⇐). Conditions 2 through 4, above, impose ordering constraints on some pairs of nonoverlapping read/write accesses.

Table 5–1 permits a read access Pi:R<n>(y,b) to be ordered BEFORE an overlapping write access Pi:W<m>(x,a) that precedes the read access in processor issue order. This asymmetry for reads allows reads to be satisfied by using data from an earlier write in processor issue sequence by the same processor (for example, by hitting in a write buffer) before the write completes. The write access remains "visible" to the read access; "visibility" is described in Sections 5.6.1.5 and 5.6.1.6 and illustrated in Litmus Test 11 in Section 5.6.2.11.

An I-fetch Pi:I<4>(y,b) may also be ordered BEFORE an overlapping write Pi:W<m>(x,a) that precedes it in processor issue sequence. In that case, the write may, but need not, be visible to the I-fetch. This asymmetry in Table 5–1 allows writes to the I-stream to be incoherent until a CALL_PAL IMB is executed.

Implementations are free to perform memory accesses from a single processor in any sequence that is consistent with processor issue constraints.

5.6.1.4 Definition of Location Access Constraints

Location access constraints are imposed on overlapping read/write accesses. If *u* and *v* are overlapping read/write accesses, at least one of which is a write, then *u* and *v* must be comparable in the BEFORE (⇐) ordering, that is, either u ⇐ v or v ⇐ u.

There is no direct requirement that nonoverlapping accesses be comparable in the BEFORE (⇐) ordering.

All writes accessing any given byte are totally ordered, and any read or I-fetch accessing a given byte is ordered with respect to all writes accessing that byte.

5.6.1.5 Definition of Visibility

If *u* is a write access Pi:W<m>(x,a) and *v* is an overlapping read access Pj:R<n>(y,b), *u* is visible to *v* only if:

u ⇐ *v*, or

u precedes *v* in processor issue sequence (possible only if Pi=Pj).

If *u* is a write access Pi:W<m>(x,a) and *v* is an overlapping instruction fetch Pj:I<4>(y,b), there are the following rules for visibility:

1. If *u* ⇐ *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 Pi:W<m>(x,a), and *v* is either Pj:I<4>(y,b) or Pj:R<n>(y,b), and z is a byte accessed by both *u* and *v*, and *u* is visible to *v*; and there is no write that is AFTER *u*, is visible to *v*,

and accesses byte z; then the value of byte z read by *v* is exactly the value written by *u*. 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 Pi to Pj is modeled as Pi writing to a location INTij, and Pj reading from INTij.)

5.6.1.7 Definition of Dependence Constraint

The depends relation (DP) is defined as follows. Given *u* and *v* issued by processor Pi, where *u* is a read or an instruction fetch and *v* is a write, *u* precedes *v* in DP order (written *u* DP *v*, that is, *v* depends on *u*) in either of the following situations:

- *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 (⇐) 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 R1, …, R*n*, and writes W1, …, W*n*, such that:

1. $n \geq 1$,
2. For each *i*, $1 \leq i \leq n$, Ri DP Wi,
3. For each *i*, $1 \leq i < n$, Wi is a source of Ri + 1, and
4. W*n* is a source of R1.

That constraint eliminates the possibility of "causal loops." A simple example of a "causal loop" is when the execution of a write on Pi depends on the execution of a write on Pj and vice versa, creating a circular dependence chain. The following simple example of a "causal loop" is written in the style of the litmus tests in Section 5.6.2, where initially *x* and *y* are 1:

Processor Pi executes:

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

Processor Pj executes:

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

Representing those code sequences in the style of the litmus tests in Section 5.6.2, it is impossible for the following sequence to result:

Pi	Pj
[U1] Pi:R<8>(x,0)	[V1] Pj:R<8>(y,0)
[U2] Pi:W<8>(y,0)	[V2] Pj: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 Pi's program:

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

Processor Pj's program:

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

5.6.1.8 Definition of Load-Locked and Store-Conditional

The property of load-locked and store-conditional applies only to memory-like regions.

For each successful store-conditional *v*, there exists a load-locked *u* such that the following are true:

1. *u* precedes *v* in the processor issue sequence.
2. There is no load-locked or store-conditional between *u* and *v* in the processor issue sequence.
3. If *u* and *v* access within the same naturally aligned 16-byte physical and virtual block in memory, then for every write *w* by a different processor that accesses within *u*'s lock range (where *w* is either a store or a successful store conditional), it must be true that *w* $\Leftarrow$ 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 ⇐ order) write to *x* visible to V2.

<3> From <1> and <2>, V1 ⇐ U1.

<4> Since U1 is visible to V2, and they are issued by different processors, U1 ⇐ V2.

<5> By the processor issue constraints, V2 ⇐ V3.

<6> From <4> and <5>, U1 ⇐ 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 ⇐ V1.

<2> W1 reading 3 implies V1 is visible to W1, so V1 ⇐ W1 ⇐ 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 ⇐ U1.

<4> From <1> and <3>, U1 ⇐ V1 ⇐ 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 ⇐ V1, by storage and visibility.

<2> Since V2 does not read 2, there cannot be U1 ⇐ V2.

<3> By the access order constraints, it follows from <2> that V2 ⇐ U1.

There are no conflicts in the sequence. There are no violations of the definition of BEFORE.

5.6.2.5 Litmus Test 5 (Sequence Okay)

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 ⇐ V1, by storage and visibility.

<2> V1 ⇐ V2 ⇐ V3, by processor issue constraints.

<3> V3 reading 1 implies V3 ⇐ U1, by storage and visibility.

There is U2 ⇐ V1 ⇐ V2 ⇐ V3 ⇐ U1. There are no conflicts in this sequence. There are no violations of the definition of BEFORE.

5.6.2.6 Litmus Test 6 (Sequence Okay)

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 ⇐ U2 ⇐ U3, by processor issue constraints.

<2> V1 reading 2 implies U3 ⇐ V1, by storage and visibility.

<3> V2 reading 1 implies V2 ⇐ U1, by storage and visibility.

There is V2 ⇐ U1 ⇐ U2 ⇐ U3 ⇐ V1. There are no conflicts in this sequence. There are no violations of the definition of BEFORE.

In litmus tests 4, 5, and 6, writes to two different locations *x* and *y* are observed (by another processor) to occur in the opposite order than that in which they were performed. An update to *y* propagates quickly to Pj, but the update to *x* is delayed, and Pi and Pj 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 ⇐ U1, by storage and visibility.

<2> V1 reading 2 implies U3 ⇐ V1, by storage and visibility.

<3> U1 ⇐ U2 ⇐ U3, by processor issue constraints.

<4> V1 ⇐ V2 ⇐ V3, by processor issue constraints.

<5> By <2>, <3>, and <4>, U1 ⇐ U2 ⇐ U3 ⇐ V1 ⇐ V2 ⇐ 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 ⇐ U1, by storage and visibility.

<2> U3 reading 1 implies U3 ⇐ V1, by storage and visibility.

<3> U1 ⇐ U2 ⇐ U3, by processor issue constraints.

<4> V1 ⇐ V2 ⇐ V3, by processor issue constraints.

<5> By <2>, <3>, and <4>, U1 ⇐ U2 ⇐ U3 ⇐ V1 ⇐ V2 ⇐ 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 equivalent[1] if it is a DMA I/O device), then write a flag (equivalently, send an interrupt) signaling the other processor that the shared data is ready. Each receiving processor must read the new flag (equivalently, receive the interrupt), execute an MB, then read or update the shared data. In the special case in which data

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.

is communicated through just one location in memory, memory barriers are not necessary.

Software Note:

Note that this section does not describe how to reliably communicate data from a processor to a DMA device. See Section 5.6.4.7.

Leaving out the first MB removes the assurance that the shared data is written before the flag is written.

Leaving out the second MB removes the assurance that the shared data is read or updated only after the flag is seen to change; in this case, an early read could see an old value, and an early update could be overwritten.

This implies that after a DMA I/O device has written some data to memory (such as paging in a page from disk), the DMA device must logically execute an MB[1] before posting a completion interrupt, and the interrupt handler software must execute an MB before the data is guaranteed to be visible to the interrupted processor. Other processors must also execute MBs before they are guaranteed to see the new data.

An important special case occurs when a write is done (perhaps by an I/O device) to some physical page frame, then an MB is executed, and then a previously invalid PTE is changed to be a valid mapping of the physical page frame that was just written. In this case, all processors that access virtual memory by using the newly valid PTE must guarantee to deliver the newly written data after the TB miss, for both I-stream and D-stream accesses.

5.6.4.4 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.

1 See Footnote 1 on page 5-22.

This implies that after a DMA I/O device has written some I-stream to memory (such as paging in a page from disk), the DMA device must logically execute an MB[1] 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.

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.

1 See Footnote 1 on page 5-22.

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.	⇒	:
:		Pickup 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		
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 MB[1]
3. Read the shared data.

As shown in Section 5.6.4.3, leaving out the memory barrier removes the assurance that the shared data is written before the flag is. Unlike the case in Section 5.6.4.3, writing the shared data to a non-memory-like physical memory region removes the assurance that the I/O device

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.

will detect the writes of the shared data before detecting the flag write, interrupt, or device register write.

This implies that after a processor has prepared a data buffer to be read from memory by a DMA I/O device (such as writing a buffer to disk), the processor must execute an MB before starting the I/O. The I/O device, after receiving the start signal, must logically execute an MB before reading the data buffer, and the buffer must be located in a memory-like physical memory region.

There are methods of communicating data that may work in some systems but are not guaranteed in all systems. Two notable examples are:

1. If an Alpha processor writes a location implemented in a component located on an I/O bus in the system, then executes a memory barrier, then writes a flag in some memory location (in a memory-like or non-memory-like region), a device 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.

Chapter 6

Common PALcode Architecture

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.

For example, in a MOS VLSI implementation, a small (32-entry) fully associative TB can be the right match to the media, given that chip area is a costly resource. In an ECL version, a large (1024 entry) direct-mapped TB can be used because it will use RAM chips and does not have fast associative memories available. This difference would be handled by implementation-specific versions of the PALcode on the two systems, both versions providing transparent TB miss service routines. The operating system code would not need to know there were any differences.

An Alpha Privileged Architecture Library (PALcode) of routines and environments is supplied by 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 Section C.11 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

Mnemonic	Name
BPT	Breakpoint trap
BUGCHK	Bugcheck trap
CSERVE	Console service
GENTRAP	Generate trap
RDUNIQUE	Read unique value
SWPPAL	Swap PALcode
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

Mnemonic	Type	Operation
DRAINA	Privileged	Drain aborts
HALT	Privileged	Halt processor
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 Console Interface (III), Chapter 3, in the *Alpha Architecture Reference Manual*.

6.7.3 Instruction Memory Barrier

Format:

```
CALL_PAL        IMB                          !PALcode format
```

Operation:

```
{Make instruction stream coherent with data stream}
```

Exceptions:

None

Instruction mnemonics:

```
CALL_PAL IMB                 I-stream Memory Barrier
```

Description:

An IMB instruction must be executed after software or I/O devices write into the instruction stream or modify the instruction stream virtual address mapping, and before the new value is fetched as an instruction. An implementation may contain an instruction cache that does not track either processor or I/O writes into the instruction stream. The instruction cache and memory are made coherent by an IMB instruction.

If the instruction stream is modified and an IMB is not executed before fetching an instruction from the modified location, it is UNPREDICTABLE whether the old or new value is fetched.

Software Note:

In a multiprocessor environment, executing an IMB on one processor does not affect instruction caches on other processors. Thus, a single IMB on one processor is insufficient to guarantee that all processors see a modification of the instruction stream.

The cache coherency and sharing rules are described in Console Interface (III), Chapter 2, in the *Alpha Architecture Reference Manual*.

Chapter 7

Console Subsystem Overview

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.

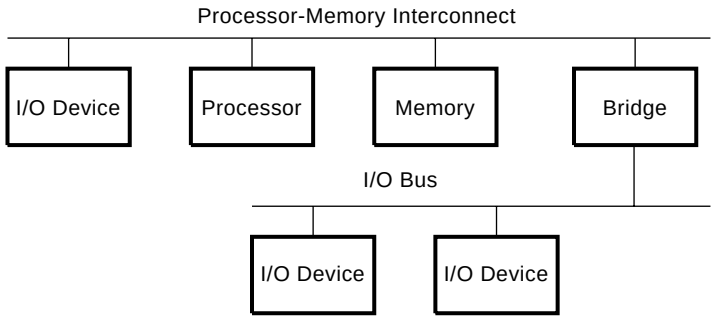
Chapter 8

Input/Output Overview

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

Chapter 9

OpenVMS Alpha

The following sections specify the Privileged Architecture Library (PALcode) instructions, that are required to support an OpenVMS Alpha system.

9.1 Unprivileged OpenVMS Alpha PALcode

The unprivileged PALcode instructions provide support for system operations to all modes of operation (kernel, executive, supervisor, and user).

Table 9–1 describes the unprivileged OpenVMS Alpha PALcode instructions.

Table 9–1 : Unprivileged OpenVMS Alpha PALcode Instruction Summary

Mnemonic	**Operation and Description**
BPT	Breakpoint The BPT instruction is provided for program debugging. It switches the processor to kernel mode and pushes R2..R7, the updated PC, and PS on the kernel stack. It then dispatches to the address in the Breakpoint vector, stored in a control block.
BUGCHK	Bugcheck The BUGCHK instruction is provided for error reporting. It switches the processor to kernel mode and pushes R2..R7, the updated PC, and PS on the kernel stack. It then dispatches to the address in the bugcheck vector, stored in a control block. The value in R16 specifies the particular bugcheck type.

Table 9–1 : Unprivileged OpenVMS Alpha PALcode Instruction Summary (Continued)

Mnemonic	**Operation and Description**
CHME	Change mode to executive The CHME instruction allows a process to change its mode in a controlled manner. A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. Registers R2..R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHME instruction. The value in R16 specifies the particular exception type.
CHMK	Change mode to kernel CHMK allows a process to change its mode to kernel in a controlled manner. A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PS, and PC are pushed onto the kernel stack. The saved PC addresses the instruction following the CHMK instruction. The value in R16 specifies the particular exception type.
CHMS	Change mode to supervisor CHMS allows a process to change its mode in a controlled manner. A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHMS instruction. The value in R16 specifies the particular exception type.
CHMU	Change mode to user CHMU allows a process to call a routine via the change mode mechanism.R2..R7, PS, and PC are pushed onto the current stack. The saved PC addresses the instruction following the CHMU instruction. The value in R16 specifies the particular exception type.
CLRFEN	Clear floating-point enable CLRFEN writes a zero to the floating-point enable register.
GENTRAP	Generate trap GENTRAP is provided for reporting runtime software conditions. It switches the processor to kernel mode and pushes registers R2..R7, the updated PC, and the PS on the kernel stack. It then dispatches to the address of the GENTRAP vector, stored in a control block.

Table 9–1 : Unprivileged OpenVMS Alpha PALcode Instruction Summary (Continued)

Mnemonic	**Operation and Description**
IMB	I-Stream memory barrier IMB ensures that the contents of an instruction cache are coherent after the instruction stream has been modified by software or I/O devices.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.
INSQHIL	Insert into longword queue at header, interlocked The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.
INSQHILR	Insert into longword queue at header, interlocked resident The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.
INSQHIQ	Insert into quadword queue at header, interlocked The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.
INSQHIQR	Insert into quadword queue at header, interlocked resident The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.
INSQTIL	Insert into longword queue at tail, interlocked The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.

Table 9–1 : Unprivileged OpenVMS Alpha PALcode Instruction Summary (Continued)

Mnemonic	**Operation and Description**
INSQTILR	Insert into longword queue at tail, interlocked resident The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.
INSQTIQ	Insert into quadword queue at tail, interlocked The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.
INSQTIQR	Insert into quadword queue at tail, interlocked resident The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.
INSQUEL	Insert into longword queue The entry specified in R17 is inserted into the absolute queue following the entry specified by the predecessor addressed by R16 for INSQUEL, or following the entry specified by the contents of the longword addressed by R16 for INSQUEL/D. The insertion is a noninterruptible operation.
INSQUEQ	Insert into quadword queue The entry specified in R17 is inserted into the absolute queue following the entry specified by the predecessor addressed by R16 for INSQUEQ, or following the entry specified by the contents of the quadword addressed by R16 for INSQUEQ/D. The insertion is a noninterruptible operation.
PROBE	Probe read/write access PROBE checks the read (PROBER) or write (PROBEW) accessibility of the first and last byte specified by the base address and the signed offset; the bytes in between are not checked. System software must check all pages between the two bytes if they are to be accessed. <p> PROBE is only intended to check a single datum for accessibility.

Table 9–1 : Unprivileged OpenVMS Alpha PALcode Instruction Summary (Continued)

Mnemonic	**Operation and Description**
RD_PS	Read processor status RD_PS writes the Processor Status (PS) to register R0.
READ_UNQ	Read unique context READ_UNQ reads the hardware process (thread) unique context value, if previously written by WRITE_UNQ, and places that value in R0.
REI	Return from exception or interrupt The PS, PC, and saved R2..R7 are popped from the current stack and held in temporary registers. The new PS is checked for validity and consistency. If it is valid and consistent, the current stack pointer is then saved and a new stack pointer is selected. Registers R2 through R7 are restored by using the saved values held in the temporary registers. A check is made to determine if an AST or interrupt is pending.If the enabling conditions are present for an interrupt or AST at the completion of this instruction, the interrupt or AST occurs before the next instruction.
REMQHIL	Remove from longword queue at header, interlocked The self-relative queue entry following the header, pointed to by R16, is removed from the queue, and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
REMQHILR	Remove from longword queue at header, interlocked resident The queue entry following the header, pointed to by R16, is removed from the self-relative queue, and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation. This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.
REMQHIQ	Remove from quadword queue at header, interlocked The self-relative queue entry following the header, pointed to by R16, is removed from the queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.

Table 9–1 : Unprivileged OpenVMS Alpha PALcode Instruction Summary (Continued)

Mnemonic	**Operation and Description**
REMQHIQR	Remove from quadword queue at header, interlocked resident The queue entry following the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation. This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.
REMQTIL	Remove from longword queue at tail, interlocked The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
REMQTILR	Remove from longword queue at tail, interlocked resident The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation. This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.
REMQTIQ	Remove from quadword queue at tail, interlocked The self-relative queue entry preceding the header, pointed to by R16, is removed from the queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
REMQTIQR	Remove from quadword queue at tail, interlocked resident The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation. This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.

Table 9–1 : Unprivileged OpenVMS Alpha PALcode Instruction Summary (Continued)

Mnemonic	**Operation and Description**
REMQUEL	Remove from longword queue The queue entry addressed by R16 for REMQUEL or the entry addressed by the longword addressed by R16 for REMQUEL/D is removed from the longword absolute queue, and the address of the removed entry is returned in R1. The removal is a noninterruptible operation.
REMQUEQ	Remove from quadword queue The queue entry addressed by R16 for REMQUEQ or the entry addressed by the quadword addressed by R16 for REMQUEL/D is removed from the quadword absolute queue, and the address of the removed entry removed is returned in R1. The removal is a noninterruptible operation.
RSCC	Read system cycle counter Register R0 is written with the value of the system cycle counter. This counter is an unsigned 64-bit integer that increments at the same rate as the process cycle counter. The system cycle counter is suitable for timing a general range of intervals to within 10% error and may be used for detailed performance characterization.
SWASTEN	Swap AST enable SWASTEN swaps the AST enable bit for the current mode. The new state for the enable bit is supplied in register R16<0> and previous state of the enable bit is returned, zero-extended, in R0. A check is made to determine if an AST is pending. If the enabling conditions are present for an AST at the completion of this instruction, the AST occurs before the next instruction.
WRITE_UNQ	Write unique context WRITE_UNQ writes the hardware process (thread) unique context value passed in R16 to internal storage or to the hardware privileged context block.
WR_PS_SW	Write processor status software field WR_PS_SW writes the Processor Status software field (PS<SW>) with the low-order three bits of R16<2:0>.

9.2 Privileged OpenVMS Alpha Palcode

The privileged PALcode instructions can be called in kernel mode only.

Table 9–2 describes the privileged OpenVMS Alpha PALcode instructions.

Table 9–2 : Privileged OpenVMS Alpha PALcode Instructions Summary

Mnemonic	Operation and Description
CFLUSH	Cache flush At least the entire physical page specified by a page frame number in R16 is flushed from any data caches associated with the current processor. After doing a CFLUSH, the first subsequent load on the same processor to an arbitrary address in the target page is fetched from physical memory.
CSERVE	Console service CSERVE is specific to each PALcode and console implementation and is not intended for operating system use.
DRAINA	Drain aborts DRAINA stalls instruction issuing until all prior instructions are guaranteed to complete without incurring aborts.
HALT	Halt processor HALT stops normal instruction processing.
LDQP	Load quadword physical The quadword-aligned memory operand, whose physical address is in R16, is fetched and written to R0. If the operand address in R16 is not quadword-aligned, the result is UNPREDICTABLE.
MFPR	Move from processor register The internal processor register specified by the PALcode function field is written to R0.
MTPR	Move to processor register The source operands in integer registers R16 (and R17, reserved for future use) are written to the internal processor register specified by the PALcode function field. The effect of loading a processor register is guaranteed to be active on the next instruction.

Table 9–2 : Privileged OpenVMS Alpha PALcode Instructions Summary (Continued)

Mnemonic	Operation and Description
STQP	Store quadword physical The quadword contents of R17 are written to the memory location whose physical address is in R16. If the operand address in R16 is not quadword-aligned, the result is UNPREDICTABLE.
SWPCTX	Swap privileged context SWPCTX returns ownership of the data structure that contains the current hardware privileged context (the HWPCB) to the operating system and passes ownership of the new HWPCB to the processor.
SWPPAL	Swap PALcode image SWPPAL causes the current PALcode to be replaced by the specified new PALcode image. Intended for use by operating systems only during bootstraps and by consoles during transitions to console I/O mode.
WTINT	Wait for interrupt 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; or, the first clock tick after a specified number of clock ticks has been skipped.

Chapter 10

Digital UNIX

The following sections specifiy the Privileged Architecture Library (PALcode) instructions that are required to support a Digital UNIX system.

10.1 Unprivileged Digital UNIX PALcode

Table 10–1 describes the unprivileged Digital UNIX PALcode instructions.

Table 10–1 : Unprivileged Digital UNIX PALcode Instruction Summary

Mnemonic	Operation and Description
bpt	Break point trap The bpt instruction switches mode to kernel, builds a stack frame on the kernel stack, and dispatches to the breakpoint code.
bugchk	Bugcheck The bugchk instruction switches mode to kernel, builds a stack frame on the kernel stack, and dispatches to the breakpoint code.
callsys	System call The callsys instruction switches mode to kernel, builds a callsys stack frame, and dispatches to the system call code.
clrfen	Clear floating-point enable The clrfen instruction writes a zero to the floating-point enable register.
gentrap	Generate trap The gentrap instruction switches mode to kernel, builds a stack frame on the kernel stack, and dispatches to the gentrap code.
imb	I-stream memory barrier The imb instruction makes the I-cache coherent with main memory.

Table 10–1 : Unprivileged Digital UNIX PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
rdunique	Read unique The rdunique instruction returns the process unique value.
urti	Return from user mode trap The urti instruction pops from the user stack the registers a0 through a2, the global pointer, the new user assembler temporary register, the stack pointer, the program counter, and the processor status register.
wrunique	Write unique The wrunique instruction sets the process unique register.

10.2 Privileged Digital UNIX PALcode

The privileged PALcode instructions can be called only from kernel mode. They provide an interface to control the privileged state of the machine.

Table 10–2 describes the privileged Digital UNIX PALcode instructions.

Table 10–2 : Privileged Digital UNIX PALcode Instruction Summary

Mnemonic	Operation and Description
cflush	Cache flush The cflush instruction flushes an entire physical page pointed to by the specified page frame number (PFN) from any data caches associated with the current processor. All processors must implement this instruction.
cserve	Console service This instruction is specific to each PALcode and console implementation and is not intended for operating system use.
draina	Drain aborts The draina instruction stalls instruction issuing until all prior instructions are guaranteed to complete without incurring aborts.
halt	Halt processor The halt instruction stops normal instruction processing. Depending on the halt action setting, the processor can either enter console mode or the restart sequence.
rdmces	Read machine check error summary The rdmces instruction returns the MCES register in v0.
rdps	Read processor status The rdps instruction returns the current PS.
rdusp	Read user stack pointer The rdusp instruction reads the user stack pointer while in kernel mode and returns it.
rdval	Read system value The rdval instruction reads a 64-bit per-processor value and returns it.

Table 10–2 : Privileged Digital UNIX PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
retsys	Return from system call The retsys instruction pops the return address, the user stack pointer, and the user global pointer from the kernel stack. It then saves the kernel stack pointer, sets mode to user, enables interrupts, and jumps to the address popped off the stack.
rti	Return from trap, fault or interrupt The rti instruction pops certain registers from the kernel stack. If the new mode is user, the kernel stack is saved and the user stack restored.
swpctx	Swap privileged context The swpctx instruction saves the current process data in the current process control block (PCB). Then swpctx switches to the PCB and loads the new process context.
swpipl	Swap IPL The swpipl instruction returns the current value IPL and sets the IPL.
swppal	Swap PALcode image The swppal instruction causes the current PALcode to be replaced by the specified new PALcode image. Intended only for use by operating systems during bootstraps and by consoles during transitions to console I/O mode.
tbi	TB invalidate The tbi instruction removes entries from the instruction and data translation buffers when the mapping entries change.
whami	Who_Am_I The whami instruction returns the processor number for the current processor. The processor number is in the range 0 to the number of processors minus one (0..numproc–1) that can be configured in the system.
wrent	Write system entry address The wrent instruction sets the virtual address of the system entry points.
wrfen	Write floating-point enable The wrfen instruction writes a bit to the floating-point enable register.
wripir	Write interprocessor interrupt request The wripr instruction generates an interprocessor interrupt on the processor number passed as an input parameter. The interrupt request is recorded on the target processor and initiated when the proper enabling conditions are present.
wrkgp	Write kernel global pointer The wrkgp instruction writes the kernel global pointer internal register.
wrmces	Write machine check error summary The wrmces instructions clears the machine check in progress bit and clears the processor- or system-correctable error in progress bit in the MCES register. The instruction also sets or clears the processor- or system-correctable error reporting enable bit in the MCES register.
wrperfmon	Performance monitoring function The wrperfmon instruction alerts any performance monitoring software/hardware in the system to monitor the performance of this process.
wrusp	Write user stack pointer The wrusp instruction writes a value to the user stack pointer while in kernel mode.

Table 10–2 : Privileged Digital UNIX PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
wrval	Write system value The wrval instruction writes a 64-bit per-processor value.
wrvptptr	Write virtual page table pointer The wrvptptr instruction writes a pointer to the virtual page table pointer (vptptr).
wtint	Wait for interrupt The wtint instruction requests that, if possible, the PALcode wait for the first of either of the following conditions before returning: any interrupt other than a clock tick; or, the first clock tick after a specified number of clock ticks has been skipped.

Chapter 11

Windows NT Alpha

The following sections specify the Privileged Architecture Library (PALcode) instructions that are required to support a Windows NT Alpha system.

11.1 Unprivileged Windows NT Alpha PALcode

The unprivileged PALcode instuctions provide support for system operations and may be called from both kernel and user modes.

Table 11–1 : Unprivileged Windows NT Alpha PALcode Instruction Summary

Mnemonic	Operation and description
bpt	Breakpoint trap (standard user-mode breakpoint) The bpt instruction raises a breakpoint general exception to the kernel, setting a USER_BREAKPOINT breakpoint type.
callkd	Call kernel debugger The callkd instruction raises a breakpoint general exception to the kernel, setting the breakpoint type with the value supplied as an input parameter.
callsys	System service call The callsys instruction raises a system service call exception to the kernel. Callsys switches to kernel mode if necessary, builds a trap frame on the kernel stack, and then enters the kernel at the kernel system service exception handler.
gentrap	Generate a trap The gentrap instruction generates a software general exception that raises an exception code to the current thread. The exception code is generated from a trap number that is specified as an input parameter. Gentrap is used to raise software-detected exceptions such as bound check errors or overflow conditions.

Table 11–1 : Unprivileged Windows NT Alpha PALcode Instruction Summary

Mnemonic	Operation and description
imb	Instruction memory barrier The imb instruction guarantees that all subsequent instruction stream fetches are coherent with respect to main memory. Imb must be issued before executing code in memory that has been modified (either by stores from the processor or DMA from an I/O processor). User-mode code that modifies the I-stream must call the appropriate Windows NT API to ensure I-cache coherency.
kbpt	Kernel breakpoint trap The kbpt instruction raises a breakpoint general exception to the kernel, setting a KERNEL_BREAKPOINT breakpoint type.
rdteb	Read thread environment block pointer The rdteb instruction returns the contents of the TEB internal processor register for the currently executing thread (the base address of the thread environment block).

11.2 Privileged Windows NT Alpha PALcode

The privileged PALcode instuctions provide support for system operations and may be called from only kernel mode.

Table 11–2 : Privileged Windows NT Alpha PALcode Instruction Summary

Mnemonic	Operation and description
csir	Clear software interrupt request The csir instruction clears the specified bit in the SIRR internal processor register.
dalnfix	Disable alignment fixups The dalnfix instruction disables alignment 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.
di	Disable all interrupts The di instruction disables all interrupts by clearing the interrupt enable (IE) bit in the PSR internal processor register. The IRQL field is unaffected. Interrupts may be re-enabled via the ei instruction.

Table 11–2 : Privileged Windows NT Alpha PALcode Instruction Summary (Continued)

Mnemonic	Operation and description
draina	Drain all aborts including machine checks The draina instruction drains all aborts, including machine checks, from the current processor. Draina guarantees that no abort is signaled for an instruction issued before the draina while any instruction issued subsequent to the draina is executing.
dtbis	Data translation buffer invalidate single The dtbis instruction invalidates a single data stream translation. The translation for the virtual address must be invalidated in all data translation buffers and in all virtual data caches.
ealnfix	Enable alignment fixups The ealnfix instruction enables alignment 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.
ei	Enable interrupts The ei instruction enables interrupts for the IRQL set in the PSR internal processor register by setting the interrupt enable (IE) bit in the PSR.
halt	Halt the operating system by forcing illegal instruction trap The halt instruction forces an illegal instruction exception.
initpal	Initialize PALcode data structures with operating system values The initpal instruction is called early in the kernel initialization sequence to establish values for internal processor registers (IPRs) that are needed for trap and fault handling. The KGP and PCR registers are initialized once and persist throughout the run time of the operating system.
initpcr	Initialize processor control region data The initpcr instruction caches process-specific information, including parts of the interrupt level table (ILT), for use by the PALcode.
rdcounters	Read the software event counters The rdcounters instruction is only used with debug PALcode. With production PALcode, rdcounters returns a status value of zero, indicating that it is not implemented in the current PALcode image.

Table 11–2 : Privileged Windows NT Alpha PALcode Instruction Summary (Continued)

Mnemonic	Operation and description
rdirql	Read the current IRQL from the PSR The rdirql instruction returns the contents of the interrupt request level (IRQL) field of the PSR internal processor register.
rdksp	Read initial kernel stack pointer for the current thread The rdksp instruction returns the contents of the IKSP (initial kernel stack pointer) internal processor register for the currently executing thread.
rdmces	Read the machine check error summary register The rdmces instruction returns the contents of the machine check error summary (MCES) internal processor register.
rdpcr	Read the processor control region base address The rdpcr instruction returns the contents of the PCR internal processor register (the base address value of the processor control region).
rdpsr	Read the current processor status register (PSR) The rdpsr instruction returns the contents of the current PSR (Processor Status Register) internal processor register.
rdstate	Read the current internal processor state The rdstate instruction returns the internal processor state to an internal buffer.
rdthread	Read the thread value for the current thread The rdthread instruction returns the contents of the THREAD internal processor register (the value of the currently executing thread).
reboot	Transfer to console firmware The reboot instruction stops the operating system from executing and returns execution to the boot environment. Reboot is responsible for completing the ARC restart block before returning to the boot environment.
restart	Restart the operating system from the restart block The restart instruction restores saved processor state and resumes execution of the operating system.

Table 11–2 : Privileged Windows NT Alpha PALcode Instruction Summary (Continued)

Mnemonic	Operation and description
retsys	Return from system service call exception The retsys instruction returns from a system service call exception by unwinding the trap frame and returning to the code stream that was executing when the original exception was initiated. In addition, retsys accepts a parameter to set software interrupt requests that became pending while the exception was handled.
rfe	Return from trap or interrupt The rfe instruction returns from exceptions by unwinding the trap frame and returning to the code stream that was executing when the original exception was initiated. In addition, rfe accepts a parameter to set software interrupt requests that became pending while the exception was handled.
ssir	Set software interrupt request The ssir instruction sets software interrupt requests by setting the appropriate bits in the SIRR internal processor register.
swpctx	Swap thread context The swpctx instruction swaps the privileged portions of thread context. Thread context is swapped by establishing the new IKSP, THREAD, and TEB internal processor register values.
swpirql	Swap the current interrupt request level The swpirql instruction swaps the current IRQL field in the PSR internal processor register by setting the processor so that only permitted interrupts are enabled for the new IRQL. Swpirql updates the IRQL field and returns the previous IRQL.
swpksp	Swap the initial kernel stack pointer for the current thread The swpksp instruction returns the value of the previous IKSP internal processor register and writes a new IKSP for the currently executing thread.
swppal	Swap the currently executing PALcode The swppal instruction swaps the currently executing PALcode by transferring to the base address of the new PALcode image in the PALcode environment.
swpprocess	Swap process context (swap address space) The swpprocess instruction swaps the privileged process context by changing the address space for the currently executing thread.

Table 11–2 : Privileged Windows NT Alpha PALcode Instruction Summary (Continued)

Mnemonic	Operation and description
tbia	Translation buffer invalidate all The tbia instruction invalidates all translations and virtual cache blocks within the processor.
tbim	Translation buffer invalidate multiple The tbim instruction invalidates multiple virtual translations for the current ASN. The translation for the virtual address must be invalidated in all processor translation buffers and virtual caches.
tbimasn	Translation buffer invalidate multiple for ASN The tbimasn instruction invalidates multiple virtual translations for a specified ASN. The translation for the virtual addresses must be invalidated in all processor translation buffers and virtual caches.
tbis	Translation buffer invalidate single The tbis instruction invalidates a single virtual translation. The translation for the passed virtual address must be invalidated in all processor translation buffers and virtual caches.
tbisasn	Translation buffer invalidate single for ASN The tbisasn instruction invalidates a single virtual translation for a specified address space. The translation for the passed virtual address must be invalidated in all processor translation buffers and virtual caches.
wrentry	Write kernel exception entry routine The wrentry instruction provides the registry of exception handling routines for the exception classes. The kernel must use wrentry to register an exception handler for each of the exception classes.
wrmces	Write the machine check error summary register The wrmces instruction writes new values for the MCES internal processor register and returns the previous values of that register.
wrperfmon	Write performance counter interrupt control information The wrperfmon instruction writes control information for the two processor performance counters. One parameter identifies the selected performance counter, while another controls whether the selected performance counter is enabled or disabled. The instruction returns the previous enable state for the selected performance counter.

Appendix A

Software Considerations

A.1 Hardware-Software Compact

The Alpha architecture, like all RISC architectures, depends on careful attention to data alignment and instruction scheduling to achieve high performance.

Since there will be various implementations of the Alpha architecture, it is not obvious how compilers can generate high-performance code for all implementations. This chapter gives some scheduling guidelines that, if followed by all compilers and respected by all implementations, will result in good performance. As such, this section represents a good-faith compact between hardware designers and software writers. It represents a set of common goals, not a set of architectural requirements. Thus, an Appendix, not a Chapter.

Many of the performance optimizations discussed below 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 PC-sampling in PCA.

Each computer architecture has a "natural word size." For the PDP-11, it is 16 bits; for VAX, 32 bits; and for Alpha, 64 bits. Other architectures also have a natural word size that varies between 16 and 64 bits. Except for very low-end implementations, ALU data paths, cache access paths, chip pin buses, and main memory data paths are all usually the natural word size.

As an architecture becomes commercially successful, high-end implementations inevitably move to double-width data paths that can transfer an *aligned* (at an even natural word address) pair of natural words in one cycle. For Alpha, this means 128-bit wide data paths will eventually be implemented. It is difficult to get much speed advantage from paired transfers unless the code being executed has instructions and data appropriately aligned on aligned octaword boundaries. Since this is difficult to retrofit to old code, the following sections sometimes encourage "over-aligning" to octaword boundaries in anticipation of high-speed Alpha implementations.

In some cases, there are performance advantages to aligning instructions or data to cache-block boundaries, or putting data whose use is correlated into the same cache block, or trying to avoid cache conflicts by not having data whose use is correlated placed at addresses that are equal modulo the cache size. Since the Alpha architecture will have many implementations, an exact cache design cannot be outlined here.

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 implementors should give first priority to executing aligned quadwords quickly. Octaword-fetch implementors should give first priority to executing aligned octawords quickly, and second priority to executing aligned quadwords quickly. Dual-issue implementations should give first priority to issuing both halves of an aligned quadword in one cycle, and second priority to buffering and issuing other combinations.

A.2.2 Branch Prediction and Minimizing Branch-Taken — Factor of 3

In many Alpha implementations, an unexpected change in I-stream address will result in about 10 lost instruction times. "Unexpected" may mean any branch-taken or may mean a mispredicted branch. In many implementations, even a correctly predicted branch to a quadword target address will be slower than straight-line code.

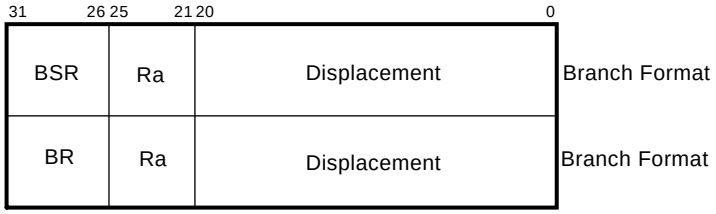
Compilers should follow these rules to minimize unexpected branches:

1. 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 whose execution profile is skewed even slightly away from 50%-50% (51-49 is enough), put the infrequent case completely out of line, so that the frequent case encounters *zero* branch-takens, and the infrequent case encounters *two*

branch-takens. If the infrequent case is rare (5%), put it far enough away that it never comes into the I-cache. If the infrequent case is extremely rare (error message code), put it on a page of rarely executed code and expect that page *never* to be paged in.

4. There are two functionally identical branch-format opcodes, BSR and BR, 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).

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 (updated PC+Instr<13:0>*4) <15:0> equals (likely_target_addr) <15:0>. In other words, pick the low 14 bits so that a normal PC+displacement*4 calculation will match the low 16 bits of the most likely target longword address. (Implementations will likely prefetch from the matching cache block.)

If the JSR is used for a computed subroutine call, compile bits <15:14> as 01, and bits <13:0> as above. Some implementations will prefetch the call target using the prediction and also push updated PC on a return-prediction stack.

If the JSR is used as a subroutine return, compile bits <15:14> as 10. Some implementations will pop an address off a return-prediction stack.

If the JSR is used as a coroutine linkage, compile bits <15:14> as 11. Some implementations will pop an address off a return-prediction stack and also push updated PC on the return-prediction stack.

Implementors should give first priority to executing straight-line code with no branch-takens as

quickly as possible, second priority to predicting conditional branches based on the sign of the displacement field (backward taken, forward not-taken), and third priority to predicting subroutine return addresses by running a small prediction stack. (VAX traces show a stack of two to four entries correctly predicts most branches.)

A.2.3 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.2.4 Instruction Scheduling — Factor of 3

The performance of Alpha programs is sensitive to how carefully the code is scheduled to minimize instruction-issue delays.

"Result latency" is defined as the number of CPU cycles that must elapse between an instruction that writes a result register and one that uses that register, if execution-time stalls are to be avoided. Thus, 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.

Compilers should try to schedule code to match the above latency rules and also to match the multiple-issue rules. If doing both is impractical for a particular sequence of code, the latency rules are more important (since they apply even in single-issue implementations).

Implementors should give first priority to minimizing the latency of back-to-back integer operations, of address calculations immediately followed by load/store, of load immediately followed by branch, and of compare immediately followed by branch. 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 128 bytes will do well for most, if not all, Alpha implementations). This applies to array results that sweep through large portions of memory, and 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.

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 will likely be a factor of 30 slower than cache hits. It is helpful to pack correlated shared data into a small number of cache blocks. It is helpful also to segregate blocks written by one processor from blocks read by others.

Therefore, accesses to shared data, including locks, should be minimized. For example, a four-processor decomposition of some manipulation of a 1000-row array should avoid accessing lock variables every row, but instead might access a lock variable every 250 rows.

Array manipulation should be partitioned across processors so that cache blocks do not thrash between processors. Having each of four processors work on every fourth array element severely impairs performance on any implementation with a cache block of four elements or larger. The processors all contend for copies of the *same* cache blocks and use only one quarter of the data in each block. Writes in one processor severely impair cache performance on all processors.

A better decomposition is to give each processor the largest possible contiguous chunk of data to work on (N/4 consecutive rows for four processors and row-major array storage; N/4 columns for column-major storage). With the possible exception of three cache blocks at the partition boundaries, this decomposition will result in each processor caching data that is touched by *no* other processor.

Operating-system scheduling algorithms should attempt to minimize process migration from one processor to another. Any time migration occurs, there are likely to be a large number of cache misses on the new processor.

Similarly, operating-system scheduling algorithms should attempt to enforce some affinity between a given device's interrupts and the processor on which the interrupt-handler runs. I/O control data structures and locks for different devices should be disjoint. 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/TB Conflicts — Factor of 1

Occasionally, programs that run with a direct-mapped cache or TB will thrash, taking excessive cache or TB misses. With some work, thrashing can be minimized at compile time.

Note:

No Alpha processor through and including the 21264 has implemented a direct-mapped TB.

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 8 KB direct-mapped cache):

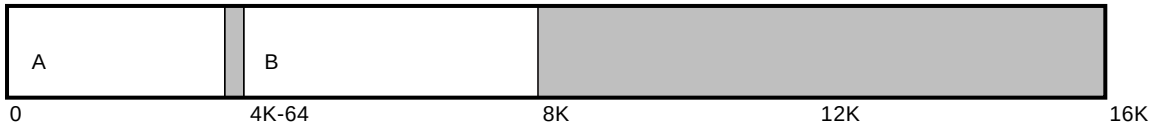
Figure A–3: Bad Allocation in Cache

BETTER allocation (A and B offset by 64 mod 2 KB, so 16 elements of A and 16 of B can be in cache simultaneously):

Figure A–4: Better Allocation in Cache

BEST allocation (A and B offset by 64 mod 2 KB, so 16 elements of A and 16 of B can be in cache simultaneously, *and* both arrays fit entirely in 8 KB or bigger cache):

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 8 KB* page, or differ in bits VA<17:13>. For loops that go through arrays in a common direction with a common stride, this requires allocating the arrays, checking that the first-itera-

tion addresses differ, and if they do not, inserting up to 8K bytes of padding *between* the arrays. This rule will avoid thrashing in direct-mapped TBs and in some large direct-mapped data caches with total sizes of 32 pages (256 KB) or more.

Usually, this padding will mean *zero* extra bytes in the executable image, just a skip in virtual address space to the next-higher page boundary.

For large caches, the rule above should be applied to the I-stream, in addition to all the D-stream references. Some implementations will have combined I-stream/D-stream large caches.

Both of the rules above can be satisfied simultaneously, thus often eliminating thrashing in all anticipated direct-mapped cache/TB implementations.

A.3.4 Sequential Read/Write — Factor of 1

All other things being equal, sequences of consecutive reads or writes should use ascending (rather than descending) memory addresses. Where possible, the memory address for a block of 2**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 Prefetching — Factor of 3

Prefetching can be directed toward a cache block (a cache line) in the primary cache.

Alpha hardware, beginning with the 21164 (EV5) and subsequent, supports cache block prefetching. Cache block prefetching is performed by the following load operations to the R31 or F31 register:

Table A–1: Cache Block Prefetching

Type	Instructions	Operation
Normal Prefetch	LDL R31, xxx (Rn)	If the load operation hits in the Dcache, the instruction is dismissed; otherwise, the addressed cache block is allocated into the Dcache.

Table A–1: Cache Block Prefetching

Type	Instructions	Operation
Prefetch with Modify Intent	LDS F31, xxx (Rn)	If the load operation hits a dirty, modified, Dcache block, the instruction is dismissed. Otherwise, the addressed cache block is allocated into the Dcache for write access — its dirty and modified bits are set.
Prefetch, Evict Next	LDQ R31, xxx (Rn)	Prefetch a cache block and mark that block in an associated cache to be evicted on the next cache fill to an associated address. (This operation is useful to prefetch data that is not to be repeatedly referenced.)

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 Section 4.6 for byte-within-register accesses are worst-case code. More importantly, they do not reflect the instructions available with the BWX extension, described in the Sections 4.2.2, 4.2.6, and 4.6.5, and in Section D.3. If the BWX extension instructions are available, it is wise to consider them rather than the sequences that follow.

The following sequences are appropriate if the BWX extension instructions are not available.

In the common case of accessing a byte or aligned word field at a known offset from a pointer that is expected to be at least longword aligned, the common-case code is much shorter. "Expected" means that the code should run fast for a longword-aligned pointer and trap for unaligned. The trap handler may at its option fix up the unaligned reference.

For access at a known offset D from a longword-aligned pointer Rx, let D.lw be D rounded down to a multiple of 4 ((D div 4)*4), and let D.mod be D mod 4.

In the common case, the intended sequence for loading and zero-extending an aligned word is:

```
LDL   R1,D.lw(Rx)            ! Traps if unaligned
EXTWL R1,#D.mod,R1           ! Picks up word at byte 0 or byte 2
```

In the common case, the intended sequence for loading and sign-extending an aligned word is:

```
LDL     R1,D.lw(Rx)          ! Traps if unaligned
SLL     R1,#48-8*D.mod,R1    ! Aligns word at high end of R1
SRA     R1,#48,R1            ! SEXT to low end of R1
```

Note:

The shifts often can be combined with shifts that might surround subsequent arithmetic operations (for example, to produce word overflow from the high end of a register).

In the common case, the intended sequence for loading and zero-extending a byte is:

```
LDL     R1,D.lw(Rx)          !
EXTBL   R1,#D.mod,R1         !
```

In the common case, the intended sequence for loading and sign-extending a byte is:

```
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 R5 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 R5 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:

```
                                ; X  = ABCD EFGH
LDG   F0,X                      ; F0 = GHEF CDAB
STT   F0,temp
LDQ   R1,temp                   ; R1 = GHEF CDAB
SLL   R1,#8,R2                  ; R2 = HEFC DAB.
SRL   R1,#8,R1                  ; R1 = .GHE FCDA
ZAP   R2,#55(hex),R2            ; R2 = H.F. D.B.
ZAP   R1,#AA(hex),R1            ; R1 = .G.E .C.A
OR    R1,R2,R1                  ; R1 = HGFE DCBA
```

For bulk swapping of arrays, this sequence can be usefully unrolled about four times and scheduled, using four different aligned quadword memory temps.

A.4.4 Stylized Code Forms

Using the same stylized code form for a common operation improves the readability of compiler output and increases the likelyhood 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 Exceptions and Trap Barriers

The EXCB instruction allows software to guarantee that in a pipelined implementation, all previous instructions have completed any behavior that is 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 distinquished 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

Table A–2: Decodable Pseudo-Operations (Stylized Code Forms) (Continued)

Pseudo-Operation in Listing		Meaning	Actual Instruction Encoding	
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
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

Table A–2: Decodable Pseudo-Operations (Stylized Code Forms) (Continued)

Pseudo-Operation in Listing		Meaning	Actual Instruction Encoding	
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 μsec of execution time, even with 100 percent cache misses. This should satisfy any requirement for a 1-msec timer interrupt rate.

Appendix B

IEEE Floating-Point Conformance

A subset of IEEE Standard for Binary Floating-Point Arithmetic (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 section 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.

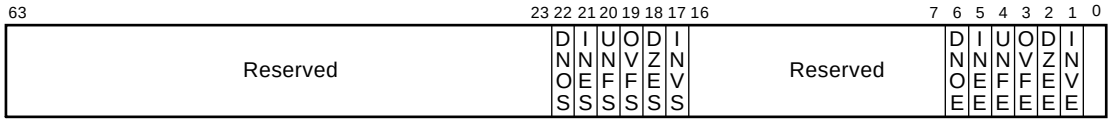
- The first option maps all denormal results to a true zero value. That option is useful for improving the performance of IEEE compliant code that does not need gradual underflow and for mixing IEEE instructions that both include and do not include the /S qualifier.
- A second option treats all denormal 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.

Table B–1: Floating-Point Control (FP_C) Quadword Bit Summary (Continued)

Bit	Description
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.
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.

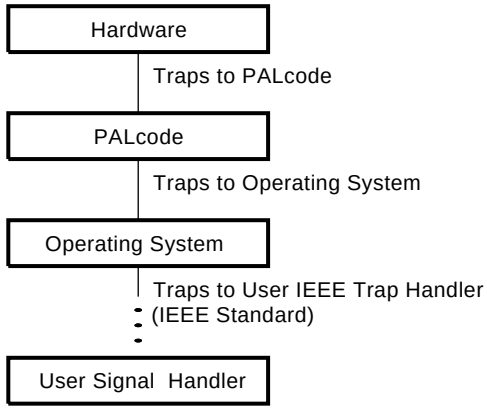
B.3 Mapping to IEEE Standard

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:

- –MAX..–MIN, –0, 0, +MIN..+MAX
- The non-finites are:
- Denormals, +/– Infinity, Signaling NaN, Quiet NaN

Alpha hardware must treat minus zero operands and results as special cases, as required by the IEEE standard.

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 4.7.8 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
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]

Table B–2: IEEE Floating-Point Trap Handling (Continued)

Alpha Instructions	Hardware[1]	PAL-Code	OS Completion Handler	User Signal Handler
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]
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]

Table B–2: IEEE Floating-Point Trap Handling (Continued)

Alpha Instructions	Hardware[1]	PAL-Code	OS Completion Handler	User Signal Handler
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
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply Cvt 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
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 ≥ 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 ≤ zero
BLT	Bra	3A	Branch if < zero
BNE	Bra	3D	Branch if ≠ 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 ≥ 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 ≤ zero
CMOVLT	Opr	11.44	CMOVE if < zero
CMOVNE	Opr	11.26	CMOVE if ≠ 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
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

Table C–2: Common Architecture Instructions (Continued)

Mnemonic	Format	Opcode	Description
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
FBGE	Bra	36	Floating branch if ≥ zero
FBGT	Bra	37	Floating branch if > zero
FBLE	Bra	33	Floating branch if ≤ zero
FBLT	Bra	32	Floating branch if < zero
FBNE	Bra	35	Floating branch if ≠ zero
FCMOVEQ	F-P	17.02A	FCMOVE if = zero
FCMOVGE	F-P	17.02D	FCMOVE if ≥ zero
FCMOVGT	F-P	17.02F	FCMOVE if > zero
FCMOVLE	F-P	17.02E	FCMOVE if ≤ zero
FCMOVLT	F-P	17.02C	FCMOVE if < zero
FCMOVNE	F-P	17.02B	FCMOVE if ≠ 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

Table C–2: Common Architecture Instructions (Continued)

Mnemonic	Format	Opcode	Description
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
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
MSKLH	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
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
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

Table C–2: Common Architecture Instructions (Continued)

Mnemonic	Format	Opcode	Description
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
WH64	Mfc	18.F800	Write hint — 64 bytes
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

C.2 IEEE Floating-Point Instructions

Table C–3 lists the hexadecimal value of the 11-bit function code field for the IEEE floating-point instructions, with and without qualifiers. The opcode for the following instructions is 16_{16}, except for SQRTS and SQRTT, which are opcode 14_{16}.

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							
CMPTLE	0A7							
CMPTLT	0A6							
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

	/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							
CMPTLE	5A7							
CMPTLT	5A6							
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

Table C–3: IEEE Floating-Point Instruction Function Codes (Continued)

	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			
CMPGLE	0A7				4A6			
CMPGLT	0A6				4A7			
CVTGD	0AD	02D	1AD	12D	4AD	42D	5AD	52D
CVTGF	0AC	02C	1AC	12C	4AC	42C	5AC	52C
CVTQF	0BC	03C						
CVTQG	0BE	03E						
CVTGQ	See below							
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		
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\	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\	LDT (mem)	LDQ_L (mem)	FBLE (br)	BLE (br)
4/C	Res	LDWU (mem)	ITFP*	FPTI*	STF (mem)	STL (mem)	BSR (br)	BLBS (br)
5/D	Res	STW (mem)	FLTV* (op)	\PAL\	STG (mem)	STQ (mem)	FBNE (br)	BNE (br)
6/E	Res	STB (mem)	FLTI* (op)	\PAL\	STS (mem)	STL_C (mem)	FBGE (br)	BGE (br)
7/F	Res	STQ_U (mem)	FLTL* (op)	\PAL\	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\	Reserved for PALcode
Res	Reserved for Compaq

C.6 Common Architecture Opcodes in Numerical Order

Table C–8: Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
00	CALL_PAL	11.26	CMOVNE	14.014	ITOFF
01	OPC01	11.28	ORNOT	14.024	ITOFT
02	OPC02	11.40	XOR	14.02A	SQRTG/C
03	OPC03	11.44	CMOVLT	14.02B	SQRTT/C
04	OPC04	11.46	CMOVGE	14.04B	SQRTS/M
05	OPC05	11.48	EQV	14.06B	SQRTT/M
06	OPC06	11.61	AMASK	14.08A	SQRTF
07	OPC07	11.64	CMOVLE	14.08B	SQRTS
08	LDA	11.66	CMOVGT	14.0AA	SQRTG
09	LDAH	11.6C	IMPLVER	14.0AB	SQRTT
0A	LDBU	12.02	MSKBL	14.0CB	SQRTS/D
0B	LDQ_U	12.06	EXTBL	14.0EB	SQRTT/D
0C	LDWU	12.0B	INSBL	14.10A	SQRTF/UC
0D	STW	12.12	MSKWL	14.10B	SQRTS/UC
0E	STB	12.16	EXTWL	14.12A	SQRTG/UC
0F	STQ_U	12.1B	INSWL	14.12B	SQRTT/UC
10.00	ADDL	12.22	MSKLL	14.14B	SQRTS/UM
10.02	S4ADDL	12.26	EXTLL	14.16B	SQRTT/UM
10.09	SUBL	12.2B	INSLL	14.18A	SQRTF/U
10.0B	S4SUBL	12.30	ZAP	14.18B	SQRTS/U
10.0F	CMPBGE	12.31	ZAPNOT	14.1AA	SQRTG/U
10.12	S8ADDL	12.32	MSKQL	14.1AB	SQRTT/U
10.1B	S8SUBL	12.34	SRL	14.1CB	SQRTS/UD
10.1D	CMPULT	12.36	EXTQL	14.1EB	SQRTT/UD
10.20	ADDQ	12.39	SLL	14.40A	SQRTF/SC
10.22	S4ADDQ	12.3B	INSQL	14.42A	SQRTG/SC
10.29	SUBQ	12.3C	SRA	14.48A	SQRTF/S
10.2B	S4SUBQ	12.52	MSKWH	14.4AA	SQRTG/S
10.2D	CMPEQ	12.57	INSWH	14.50A	SQRTF/SUC
10.32	S8ADDQ	12.5A	EXTWH	14.50B	SQRTS/SUC
10.3B	S8SUBQ	12.62	MSKLH	14.52A	SQRTG/SUC
10.3D	CMPULE	12.67	INSLH	14.52B	SQRTT/SUC
10.40	ADDL/V	12.6A	EXTLH	14.54B	SQRTS/SUM
10.49	SUBL/V	12.72	MSKQH	14.56B	SQRTT/SUM
10.4D	CMPLT	12.77	INSQH	14.58A	SQRTF/SU
10.60	ADDQ/V	12.7A	EXTQH	14.58B	SQRTS/SU
10.69	SUBQ/V	13.00	MULL	14.5AA	SQRTG/SU
10.6D	CMPLE	13.20	MULQ	14.5AB	SQRTT/SU
11.00	AND	13.30	UMULH	14.5CB	SQRTS/SUD
11.08	BIC	13.40	MULL/V	14.5EB	SQRTT/SUD
11.14	CMOVLBS	13.60	MULQ/V	14.70B	SQRTS/SUIC
11.16	CMOVLBC	14.004	ITOFS	14.72B	SQRTT/SUIC
11.20	BIS	14.00A	SQRTF/C	14.74B	SQRTS/SUIM
11.24	CMOVEQ	14.00B	SQRTS/C	14.76B	SQRTT/SUIM

Table C–8: Common Architecture Opcodes in Numerical Order (Continued)

Opcode		Opcode		Opcode	
14.78B	SQRTS/SUI	15.12F	CVTGQ/VC	15.521	SUBG/SUC
14.7AB	SQRTT/SUI	15.180	ADDF/U	15.522	MULG/SUC
14.7CB	SQRTS/SUID	15.181	SUBF/U	15.523	DIVG/SUC
14.7EB	SQRTT/SUID	15.182	MULF/U	15.52C	CVTGF/SUC
15.000	ADDF/C	15.183	DIVF/U	15.52D	CVTGD/SUC
15.001	SUBF/C	15.19E	CVTDG/U	15.52F	CVTGQ/SVC
15.002	MULF/C	15.1A0	ADDG/U	15.580	ADDF/SU
15.003	DIVF/C	15.1A1	SUBG/U	15.581	SUBF/SU
15.01E	CVTDG/C	15.1A2	MULG/U	15.582	MULF/SU
15.020	ADDG/C	15.1A3	DIVG/U	15.583	DIVF/SU
15.021	SUBG/C	15.1AC	CVTGF/U	15.59E	CVTDG/SU
15.022	MULG/C	15.1AD	CVTGD/U	15.5A0	ADDG/SU
15.023	DIVG/C	15.1AF	CVTGQ/V	15.5A1	SUBG/SU
15.02C	CVTGF/C	15.400	ADDF/SC	15.5A2	MULG/SU
15.02D	CVTGD/C	15.401	SUBF/SC	15.5A3	DIVG/SU
15.02F	CVTGQ/C	15.402	MULF/SC	15.5AC	CVTGF/SU
15.03C	CVTQF/C	15.403	DIVF/SC	15.5AD	CVTGD/SU
15.03E	CVTQG/C	15.41E	CVTDG/SC	15.5AF	CVTGQ/SV
15.080	ADDF	15.420	ADDG/SC	16.000	ADDS/C
15.081	SUBF	15.421	SUBG/SC	16.001	SUBS/C
15.082	MULF	15.422	MULG/SC	16.002	MULS/C
15.083	DIVF	15.423	DIVG/SC	16.003	DIVS/C
15.09E	CVTDG	15.42C	CVTGF/SC	16.020	ADDT/C
15.0A0	ADDG	15.42D	CVTGD/SC	16.021	SUBT/C
15.0A1	SUBG	15.42F	CVTGQ/SC	16.022	MULT/C
15.0A2	MULG	15.480	ADDF/S	16.023	DIVT/C
15.0A3	DIVG	15.481	SUBF/S	16.02C	CVTTS/C
15.0A5	CMPGEQ	15.482	MULF/S	16.02F	CVTTQ/C
15.0A6	CMPGLT	15.483	DIVF/S	16.03C	CVTQS/C
15.0A7	CMPGLE	15.49E	CVTDG/S	16.03E	CVTQT/C
15.0AC	CVTGF	15.4A0	ADDG/S	16.040	ADDS/M
15.0AD	CVTGD	15.4A1	SUBG/S	16.041	SUBS/M
15.0AF	CVTGQ	15.4A2	MULG/S	16.042	MULS/M
15.0BC	CVTQF	15.4A3	DIVG/S	16.043	DIVS/M
15.0BE	CVTQG	15.4A5	CMPGEQ/S	16.060	ADDT/M
15.100	ADDF/UC	15.4A6	CMPGLT/S	16.061	SUBT/M
15.101	SUBF/UC	15.4A7	CMPGLE/S	16.062	MULT/M
15.102	MULF/UC	15.4AC	CVTGF/S	16.063	DIVT/M
15.103	DIVF/UC	15.4AD	CVTGD/S	16.06C	CVTTS/M
15.11E	CVTDG/UC	15.4AF	CVTGQ/S	16.06F	CVTTQ/M
15.120	ADDG/UC	15.500	ADDF/SUC	16.07C	CVTQS/M
15.121	SUBG/UC	15.501	SUBF/SUC	16.07E	CVTQT/M
15.122	MULG/UC	15.502	MULF/SUC	16.080	ADDS
15.123	DIVG/UC	15.503	DIVF/SUC	16.081	SUBS
15.12C	CVTGF/UC	15.51E	CVTDG/SUC	16.082	MULS
15.12D	CVTGD/UC	15.520	ADDG/SUC	16.083	DIVS

Table C–8: Common Architecture Opcodes in Numerical Order (Continued)

Opcode		Opcode		Opcode	
16.0A0	ADDT	16.182	MULS/U	16.5A3	DIVT/SU
16.0A1	SUBT	16.183	DIVS/U	16.5A4	CMPTUN/SU
16.0A2	MULT	16.1A0	ADDT/U	16.5A5	CMPTEQ/SU
16.0A3	DIVT	16.1A1	SUBT/U	16.5A6	CMPTLT/SU
16.0A4	CMPTUN	16.1A2	MULT/U	16.5A7	CMPTLE/SU
16.0A5	CMPTEQ	16.1A3	DIVT/U	16.5AC	CVTTS/SU
16.0A6	CMPTLT	16.1AC	CVTTS/U	16.5AF	CVTTQ/SV
16.0A7	CMPTLE	16.1AF	CVTTQ/V	16.5C0	ADDS/SUD
16.0AC	CVTTS	16.1C0	ADDS/UD	16.5C1	SUBS/SUD
16.0AF	CVTTQ	16.1C1	SUBS/UD	16.5C2	MULS/SUD
16.0BC	CVTQS	16.1C2	MULS/UD	16.5C3	DIVS/SUD
16.0BE	CVTQT	16.1C3	DIVS/UD	16.5E0	ADDT/SUD
16.0C0	ADDS/D	16.1E0	ADDT/UD	16.5E1	SUBT/SUD
16.0C1	SUBS/D	16.1E1	SUBT/UD	16.5E2	MULT/SUD
16.0C2	MULS/D	16.1E2	MULT/UD	16.5E3	DIVT/SUD
16.0C3	DIVS/D	16.1E3	DIVT/UD	16.5EC	CVTTS/SUD
16.0E0	ADDT/D	16.1EC	CVTTS/UD	16.5EF	CVTTQ/SVD
16.0E1	SUBT/D	16.1EF	CVTTQ/VD	16.6AC	CVTST/S
16.0E2	MULT/D	16.2AC	CVTST	16.700	ADDS/SUIC
16.0E3	DIVT/D	16.500	ADDS/SUC	16.701	SUBS/SUIC
16.0EC	CVTTS/D	16.501	SUBS/SUC	16.702	MULS/SUIC
16.0EF	CVTTQ/D	16.502	MULS/SUC	16.703	DIVS/SUIC
16.0FC	CVTQS/D	16.503	DIVS/SUC	16.720	ADDT/SUIC
16.0FE	CVTQT/D	16.520	ADDT/SUC	16.721	SUBT/SUIC
16.100	ADDS/UC	16.521	SUBT/SUC	16.722	MULT/SUIC
16.101	SUBS/UC	16.522	MULT/SUC	16.723	DIVT/SUIC
16.102	MULS/UC	16.523	DIVT/SUC	16.72C	CVTTS/SUIC
16.103	DIVS/UC	16.52C	CVTTS/SUC	16.72F	CVTTQ/SVIC
16.120	ADDT/UC	16.52F	CVTTQ/SVC	16.73C	CVTQS/SUIC
16.121	SUBT/UC	16.540	ADDS/SUM	16.73E	CVTQT/SUIC
16.122	MULT/UC	16.541	SUBS/SUM	16.740	ADDS/SUIM
16.123	DIVT/UC	16.542	MULS/SUM	16.741	SUBS/SUIM
16.12C	CVTTS/UC	16.543	DIVS/SUM	16.742	MULS/SUIM
16.12F	CVTTQ/VC	16.560	ADDT/SUM	16.743	DIVS/SUIM
16.140	ADDS/UM	16.561	SUBT/SUM	16.760	ADDT/SUIM
16.141	SUBS/UM	16.562	MULT/SUM	16.761	SUBT/SUIM
16.142	MULS/UM	16.563	DIVT/SUM	16.762	MULT/SUIM
16.143	DIVS/UM	16.56C	CVTTS/SUM	16.763	DIVT/SUIM
16.160	ADDT/UM	16.56F	CVTTQ/SVM	16.76C	CVTTS/SUIM
16.161	SUBT/UM	16.580	ADDS/SU	16.76F	CVTTQ/SVIM
16.162	MULT/UM	16.581	SUBS/SU	16.77C	CVTQS/SUIM
16.163	DIVT/UM	16.582	MULS/SU	16.77E	CVTQT/SUIM
16.16C	CVTTS/UM	16.583	DIVS/SU	16.780	ADDS/SUI
16.16F	CVTTQ/VM	16.5A0	ADDT/SU	16.781	SUBS/SUI
16.180	ADDS/U	16.5A1	SUBT/SU	16.782	MULS/SUI
16.181	SUBS/U	16.5A2	MULT/SU	16.783	DIVS/SUI

Table C–8: Common Architecture Opcodes in Numerical Order (Continued)

Opcode		Opcode		Opcode	
16.7A0	ADDT/SUI	18.4000	MB	1F	PAL1F
16.7A1	SUBT/SUI	18.4400	WMB	20	LDF
16.7A2	MULT/SUI	18.8000	FETCH	21	LDG
16.7A3	DIVT/SUI	18.A000	FETCH_M	22	LDS
16.7AC	CVTTS/SUI	18.C000	RPCC	23	LDT
16.7AF	CVTTQ/SVI	18.E000	RC	24	STF
16.7BC	CVTQS/SUI	18.E800	ECB	25	STG
16.7BE	CVTQT/SUI	18.F000	RS	26	STS
16.7C0	ADDS/SUID	18.F800	WH64	27	STT
16.7C1	SUBS/SUID	19	PAL19	28	LDL
16.7C2	MULS/SUID	1A.0	JMP	29	LDQ
16.7C3	DIVS/SUID	1A.1	JSR	2A	LDL_L
16.7E0	ADDT/SUID	1A.2	RET	2B	LDQ_L
16.7E1	SUBT/SUID	1A.3	JSR_COROUTINE	2C	STL
16.7E2	MULT/SUID	1B	PAL1B	2D	STQ
16.7E3	DIVT/SUID	1C.00	SEXTB	2E	STL_C
16.7EC	CVTTS/SUID	1C.01	SEXTW	2F	STQ_C
16.7EF	CVTTQ/SVID	1C.30	CTPOP	30	BR
16.7FC	CVTQS/SUID	1C.31	PERR	31	FBEQ
16.7FE	CVTQT/SUID	1C.32	CTLZ	32	FBLT
17.010	CVTLQ	1C.33	CTTZ	33	FBLE
17.020	CPYS	1C.34	UNPKBW	34	BSR
17.021	CPYSN	1C.35	UNPKBL	35	FBNE
17.022	CPYSE	1C.36	PKWB	36	FBGE
17.024	MT_FPCR	1C.37	PKLB	37	FBGT
17.025	MF_FPCR	1C.38	MINSB8	38	BLBC
17.02A	FCMOVEQ	1C.39	MINSW4	39	BEQ
17.02B	FCMOVNE	1C.3A	MINUB8	3A	BLT
17.02C	FCMOVLT	1C.3B	MINUW4	3B	BLE
17.02D	FCMOVGE	1C.3C	MAXUB8	3C	BLBS
17.02E	FCMOVLE	1C.3D	MAXUW4	3D	BNE
17.02F	FCMOVGT	1C.3E	MAXSB8	3E	BGE
17.030	CVTQL	1C.3F	MAXSW4	3F	BGT
17.130	CVTQL/V	1C.70	FTOIT		
17.530	CVTQL/SV	1C.78	FTOIS		
18.0000	TRAPB	1D	PAL1D		
18.0400	EXCB	1E	PAL1E		

C.7 OpenVMS Alpha PALcode Instruction Summary

Table C–9: OpenVMS Alpha Unprivileged PALcode Instructions

Mnemonic	Opcode	Description
AMOVRM	00.00A1	Atomic move from register to memory
AMOVRR	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
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: OpenVMS Alpha 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 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
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_TBCHK	00.001A	Move from processor register TBCHK
MFPR_USP	00.0022	Move from processor register USP
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_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_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 DIGITAL UNIX PALcode Instruction Summary

Table C–11: DIGITAL 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: DIGITAL 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
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
wrval	00.0031	Write system value
wrvptptr	00.002D	Write virtual page table pointer
wtint	00.003E	Wait for interrupt

C.9 Windows NT Alpha Instruction Summary

Table C–13: 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 C–14: Windows NT Alpha Privileged PALcode instructions

Mnemonic	Opcode	Description
csir	00.000D	Clear software interrupt request
dalnfix	00.0025	Disable alignment fixups
di	00.0008	Disable interrupts
draina	00.0002	Drain aborts
dtbis	00.0016	Data translation buffer invalidate single
ealnfix	00.0024	Enable alignment fixups
ei	00.0009	Enable interrupts
halt	00.0000	Trap to illegal instruction
initpal	00.0004	Initialize the PALcode
initpcr	00.0038	Initialize processor control region data
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
swpirql	00.0006	Swap IRQL
swpksp	00.0019	Swap initial kernel stack
swppal	00.000A	Swap PALcode
swpprocess	00.0011	Swap privileged process context
swpctx	00.0010	Swap privileged thread context
ssir	00.000C	Set software interrupt request
tbia	00.0014	Translation buffer invalidate all
tbim	00.0020	Translation buffer invalidate multiple
tbimasn	00.0021	Translation buffer invalidate multiple ASN
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	00.0032	Write performance monitoring values

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 Alpha	DIGITAL UNIX	Windows NT Alpha
00.0000	00.0000	HALT	halt	halt
00.0001	00.0001	CFLUSH	cflush	restart
00.0002	00.0002	DRAINA	draina	draina
00.0003	00.0003	LDQP	—	reboot
00.0004	00.0004	STQP	—	initpal
00.0005	00.0005	SWPCTX	—	wrentry
00.0006	00.0006	MFPR_ASN	—	swpirql
00.0007	00.0007	MTPR_ASTEN	—	rdirql
00.0008	00.0008	MTPR_ASTSR	—	di
00.0009	00.0009	CSERVE	cserve	ei
00.000A	00.0010	SWPPAL	swppal	swppal
00.000B	00.0011	MFPR_FEN	—	—
00.000C	00.0012	MTPR_FEN	—	ssir
00.000D	00.0013	MTPR_IPIR	wripir	csir
00.000E	00.0014	MFPR_IPL	—	rfe
00.000F	00.0015	MTPR_IPL	—	retsys
00.0010	00.0016	MFPR_MCES	rdmces	swpctx
00.0011	00.0017	MTPR_MCES	wrmces	swpprocess
00.0012	00.0018	MFPR_PCBB	—	rdmes
00.0013	00.0019	MFPR_PRBR	—	wrmces
00.0014	00.0020	MTPR_PRBR	—	tbia
00.0015	00.0021	MFPR_PTBR	—	tbis
00.0016	00.0022	MFPR_SCBB	—	dtbis
00.0017	00.0023	MTPR_SCBB	—	tbisasn
00.0018	00.0024	MTPR_SIRR	—	rdksp
00.0019	00.0025	MFPR_SISR	—	swpksp
00.001A	00.0026	MFPR_TBCHK	—	rdpsr
00.001B	00.0027	MTPR_TBIA	—	—
00.001C	00.0028	MTPR_TBIAP	—	rdpcr
00.001D	00.0029	MTPR_TBIS	—	—
00.001E	00.0030	MFPR_ESP	—	rdthread
00.001F	00.0031	MTPR_ESP	—	—
00.0020	00.0032	MFPR_SSP	—	tbim
00.0021	00.0033	MTPR_SSP	—	tbimasn
00.0022	00.0034	MFPR_USP	—	—
00.0023	00.0035	MTPR_USP	—	—
00.0024	00.0036	MTPR_TBISD	—	ealnfix
00.0025	00.0037	MTPR_TBISI	—	dalnfix
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	—
00.002D	00.0045	—	wrvptptr	—
00.002E	00.0046	MTPR_DATFX	—	—
00.0030	00.0048	—	swpctx	rdcounters
00.0031	00.0049	—	wrval	rdstate

Table C–15: PALcode Opcodes in Numerical Order (Continued)

$Opcode_{16}$	$Opcode_{10}$	OpenVMS Alpha	DIGITAL UNIX	Windows NT Alpha
00.0032	00.0050	—	rdval	wrperfmon
00.0033	00.0051	—	tbi	—
00.0034	00.0052	—	wrent	—
00.0035	00.0053	—	swpipl	—
00.0036	00.0054	—	rdps	—
00.0037	00.0055	—	wrkgp	initpcr
00.0038	00.0056	—	wrusp	—
00.0039	00.0057	—	wrperfmon	—
00.003A	00.0058	—	rdusp	—
00.003C	00.0060	—	whami	—
00.003D	00.0061	—	retsys	—
00.003E	00.0062	WTINT	wtint	—
00.003F	00.0063	MFPR_WHAMI	rti	—
00.0080	00.0128	BPT	bpt	bpt
00.0081	00.0129	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	—
00.009F	00.0159	WRITE_UNQ	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	—	—
00.00A7	00.0167	REMQTILR	—	—

Table C–15: PALcode Opcodes in Numerical Order (Continued)

$Opcode_{16}$	$Opcode_{10}$	OpenVMS Alpha	DIGITAL UNIX	Windows NT Alpha
00.00A8	00.0168	REMQHIQR	—	—
00.00A9	00.0169	REMQTIQR	—	—
00.00AA	00.0170	GENTRAP	gentrap	gentrap
00.00AB	00.0171	—	—	rdteb
00.00AC	00.0172	—	—	kbpt
00.00AD	00.0173	—	—	callkd
00.00AE	00.0174	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

Appendix D

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)

Table D–1: Processor Type Assignments (Continued)

Major Type		Minor Type	
5 =	EV5 (21164)	0 =	Reserved (Pass 1)
		1 =	Pass 2, 2.2 (rev BA, CA)
		2 =	Pass 2.3 (rev DA, EA)
		3 =	Pass 3
		4 =	Pass 3.2
		5 =	Pass 4
6 =	EV45 (21064A)	0 =	Reserved
		1 =	Pass 1
		2 =	Pass 1.1
		3 =	Pass 2
7 =	EV56 (21164A)	0 =	Reserved
		1 =	Pass 1
		2 =	Pass 2
8 =	EV6 (21264)	0 =	Reserved
		1 =	Pass 1
		2 =	Pass 2, 2.1
		3 =	Pass 2.2
		4 =	Pass 2.3
		5 =	Pass 3
9 =	PCA56 (21164PC)	0 =	Reserved
		1 =	Pass 1

For OpenVMS Alpha and DIGITAL UNIX, the processor types are stored in the Per-CPU Slot Table (SLOT[176]), pointed to by HWRPB[160].

D.2 PALcode Variation Assignments

The PALcode variation assignments are as follows:

Table D–2: PALcode Variation Assignments

Token	PALcode Type	Summary Table
0	Console	N/A
1	OpenVMS Alpha	Console Interface (III), Chapter 3, in the *Alpha Architecture Reference Manual*.

Table D–2: PALcode Variation Assignments

Token	PALcode Type	Summary Table
2	DIGITAL UNIX	Console Interface (III), Chapter 3 in the *Alpha Architecture Reference Manual*
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.

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)

Appendix E

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 DECchip 21064, DECchip 21066, and DECchip 21068 IEEE Divide Instruction Violation

The DECchip 21064, DECchip 21066, and DECchip 21068 CPUs violate the architected handling of IEEE divide instructions DIVS and DIVT with respect to reporting Inexact Result exceptions.

Note:

The DECchip 21064A, DECchip 21066A, and DECchip 21068A CPUs are compliant and require no waiver. The DECchip 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 DECchip 21064, DECchip 21066, and DECchip 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 DECchip 21064, DECchip 21066, and DECchip 21068 do not include hardware to determine that particular exactness.

E.1.2 DECchip 21064, DECchip 21066, and DECchip 21068 Write Buffer Violation

The DECchip 21064, DECchip 21066, and DECchip 21068 CPUs can be made to violate the architecture by, under one contrived case, indefinitely delaying a buffered off-chip write.

Note:

The DECchip 21064A, DECchip 21066A, and DECchip 21068A CPUs are compliant and require no waiver. The DECchip 21164 is also compliant.

The CPUs in violation can send a buffered write off-chip 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 DECchip 21264 LDx_L/STx_C with WH64 Violation

The DECchip 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 DECchip 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.2 Implementation-Specific Functionality

The following functionality, although a documentated part of the Alpha architecture, is implemented in a manner that is specific to the particular hardware implementation.

E.2.1 DECchip 21064/21066/21068 Performance Monitoring

Note:

All functions, arguments, and descriptions in this section apply to the DECchip 21064/21064A, 21066/21066A, and 21068/21068A.

PALcode instructions control the DECchip 21064/21066/21068 on-chip performance counters. For OpenVMS Alpha, the instruction is MTPR_PERFMON; for DIGITAL UNIX and Windows NT Alpha, the instruction is wrperfmon.

The instruction arguments and results are described in the following sections. The scratch register usage is operating system specific.

Two on-chip 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, off-chip 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.2.1.2, 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 Alpha and DIGITAL UNIX, 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.2.1.1 DECchip 21064/21066/21068 Performance Monitor Interrupt Mechanism

The performance monitoring interrupt mechanism varies according to the particular operating system.

For the OpenVMS Alpha 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 DIGITAL UNIX 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 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.

For the Windows NT Alpha Operating System

When a counter overflows and interrupt enabling conditions are correct, the counter causes an interrupt to PALcode. The PALcode builds a frame on the kernel stack and dispatches to the kernel at the interrupt entry point.

E.2.1.2 Functions and Arguments for the DECchip 21064/21066/21068

The functions execute on a single (the current running) processor only and are described in Table E–1.

- The OpenVMS Alpha MTPR_PERFMON instruction is called with a function code in R16, a function-specific argument in R17, and status is returned in R0.
- The DIGITAL UNIX wrperfmon instruction is called with a function code in a0, a function specific argument in a1, and status is returned in v0.
- The Windows NT Alpha wrperfmon instruction is called with input parameters a0 through a3, as shown in Table E–1.

Table E–1: DECchip 21064/21066/21068 Performance Monitoring Functions

Function	Register Usage		Comments
Enable performance monitoring			Enable takes effect at the next IPL change
DIGITAL UNIX			
	Input:	a0 = 1	Function code
		a1 = 0	Argument
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 1	Function code
		R17 = 0	Argument
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)
Windows NT Alpha			
	Input:	a0 = 0	Select counter 0
		a0 = 1	Select counter 1
		a1 = 1	Enable selected counter
Disable performance monitoring			Disable takes effect at the next IPL change
DIGITAL UNIX			
	Input:	a0 = 0	Function code
		a1 = 0	Argument
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 0	Function code
		R17 = 0	Argument
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)

Table E–1: DECchip 21064/21066/21068 Performance Monitoring Functions (Continued)

Function	Register Usage		Comments
Windows NT Alpha			
	Input:	a0 = 0	Select counter 0
		a0 = 1	Select counter 1
		a1 = 0	Disable selected counter
Select desired events (mux_ctl)			
DIGITAL UNIX			
	Input:	a0 = 2	Function code
		a1 = mux_ctl	*mux_ctl* is the exact contents of those fields from the ICCSR register, in write format, described in Table E–2.
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 2	Function code
		R17 = mux_ctl	*mux_ctl* is the exact contents of those fields from the ICCSR register, in write format, described in Table E–2.
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)
Windows NT Alpha			
	Input:	a2 = PCMUX0	For ICCSR<PCMUX0> field when a0 = 0
		a2 = PCMUX1	For ICCSR<PCMUX1> field when a0 = 1
		a3 = PC0	For ICCSR<PC0> field when a0 = 0
		a3 = PC1	For ICCSR<PC1> field when a0 = 1
Select performance monitoring options			
DIGITAL UNIX			
	Input:	a0 = 3	Function code
		a1 = opt	Function argument *opt* is: <0> = log all processes if set <1> = log only selected if set
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)

Table E–1: DECchip 21064/21066/21068 Performance Monitoring Functions (Continued)

Function	Register Usage		Comments
OpenVMS Alpha			
	Input:	R16 = 3	Function code
		R17 = opt	Function argument *opt* is: <0> = log all processes if set <1> = log only selected if set
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)

Table E–2: DECchip 21064/21066/21068 MUX Control Fields in ICCSR Register

Bits	Option	Description	
34:32	PCMUX1	Event selection, counter 1:	
		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

Table E–2: DECchip 21064/21066/21068 MUX Control Fields in ICCSR Register (Continued)

Bits	Option	Description
11:8	PCMUX0	Event selection, counter 0:

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

Bits	Option	Description
3	PC0	Frequency setting, counter 0:

Value	Description
0	2**16 (65536) events per interrupt
1	2**12 (4096) events per interrupt

Bits	Option	Description
0	PC1	Frequency setting, counter 1:

Value	Description
0	2**12 (4096) events per interrupt
1	2**8 (256) events per interrupt

E.2.2 DECchip 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 DECchip 21164/21164PC on-chip performance counters. For OpenVMS Alpha, the instruction is MTPR_PERFMON; for DIGITAL UNIX and Windows NT Alpha, the instruction is wrperfmon.

The instruction arguments and results are described in the following sections. The scratch register usage is operating system specific.

Three on-chip 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.2.2.1 Performance Monitor Interrupt Mechanism

The performance monitoring interrupt mechanism varies according to the particular operating system.

For the OpenVMS Alpha 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 DIGITAL UNIX 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 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

For the Windows NT Alpha Operating System

When a counter overflows and interrupt enabling conditions are correct, the counter causes an interrupt to PALcode. The PALcode builds a frame on the kernel stack and dispatches to the kernel at the interrupt entry point.

E.2.2.2 Windows NT Alpha Functions and Argument

The functions for Windows NT Alpha execute on only a single (the current running) processor. The wrperfmon instruction is called with the following input registers:

Input Register	Contents (Bits)	Meaning
a0	63–0	The register in Table E–3, which contains the value to be written to the hardware PMCTR register.
a1	0	When a1 = 0, write a0 to the hardware PMCTR register. When a1 = 1, read the hardware PMCTR register. The returned PMCTR register is written to register v0.
a2	2–0	Has meaning when PCSEL1 in Table E–3 has the value 0xF. Contents are determined by processor type: **Processor** / **Contents** / **Reference** 21164 / CBOX1 / Table E–15 21164PC / PM0_MUX / Table E–17
a3	2–0	Has meaning when PCSEL2 in Table E–3 has the value 0xF. Contents are determined by processor type: **Processor** / **Contents** / **Reference** 21164 / CBOX2 / Table E–16 21164PC / PM1_MUX / Table E–18

Table E–3: Bit Summary of PMCTR Register for Windows NT Alpha

Bits	Name	Meaning
63–48	CTR0	Counter 0 value
47–32	CTR1	Counter 1 value
31	PCSEL0	Counter 0 selection: **Value** **Meaning** 0 Cycles 1 Issues
30		Must be set to one[1]
29–16	CTR2	Counter 2 value
15–14	CTL0	Counter 0 control: **Value** **Meaning** 0 Counter disable, interrupt disable 1 Counter enable, interrupt disable 2 Counter enable, interrupt at count 65536 3 Counter enable, interrupt at count 256
13–12	CTL1	Counter 1 control: **Value** **Meaning** 0 Counter disable, interrupt disable 1 Counter enable, interrupt disable 2 Counter enable, interrupt at count 65536 3 Counter enable, interrupt at count 256
11–10	CTL2	Counter 2 control: **Value** **Meaning** 0 Counter disable, interrupt disable 1 Counter enable, interrupt disable 2 Counter enable, interrupt at count 16384 3 Counter enable, interrupt at count 256

Table E–3: Bit Summary of PMCTR Register for Windows NT Alpha (Continued)

Bits	Name	Meaning	
9–8	MODE_SELECT[1]	Select modes in which to count:	
		Value	**Meaning**
		0	Count all modes
		1	Count PALmode only
		2	Count all modes except PALmode
		3	Count only user mode
7–4	PCSEL1	Counter 1 selection. See Table E–13	
3–0	PCSEL2	Counter 2 selection. See Table E–14	

[1] Windows NT Alpha uses bits 30 and 9–8 differently than as documented in the 21164 Hardware Reference Manual; it uses the processor executive mode to run user (nonprivileged) code. Therefore, bit 30 is always set to one and bits 9–8 are used to select the mode.

E.2.2.3 OpenVMS Alpha and DIGITAL UNIX Functions and Arguments

The functions execute only on a single (the current running) processor and are described in Table E–4.

The OpenVMS Alpha MTPR_PERFMON instruction is called with a function code in R16, a function-specific argument in R17, and status is returned in R0.

The DIGITAL UNIX wrperfmon instruction is called with a function code in a0, a function specific argument in a1, and status is returned in v0.

Table E–4: OpenVMS Alpha and DIGITAL UNIX Performance Monitoring Functions

Function	Register Usage		Comments
Enable performance monitoring; do not reset counters			
DIGITAL UNIX			
	Input:	a0 = 1	Function code value
		a1 = arg	Argument from Table E–5
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 1	Function code value
		R17 = arg	Argument from Table E–5
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)

Table E–4: OpenVMS Alpha and DIGITAL UNIX Performance Monitoring Functions (Continued)

Function	Register Usage		Comments
Enable performance monitoring; start the counters from zero			
DIGITAL UNIX			
	Input:	a0 = 7	Function code value
		a1 = arg	Argument from Table E–5
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 7	Function code value
		R17 = arg	Argument from Table E–5
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)
Disable performance monitoring; do not reset counters			
DIGITAL UNIX			
	Input:	a0 = 0	Function code value
		a1 = arg	Argument from Table E–6
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 0	Function code value
		R17 = arg	Argument from Table E–6
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)
Select desired events (MUX_SELECT)			
DIGITAL UNIX			
	Input:	a0 = 2	Function code value
		a1 = arg	Argument from Table E–7 or E–8
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 2	Function code value
		R17 = arg	Argument from Table E–7 or E–8
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)

Table E–4: OpenVMS Alpha and DIGITAL UNIX Performance Monitoring Functions (Continued)

Function	Register Usage		Comments
Select Processor Mode options			
DIGITAL UNIX			
	Input:	a0 = 3	Function code value
		a1 = arg	Argument from Table E–9
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 3	Function code value
		R17 = arg	Argument from Table E–9
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)
Select interrupt frequencies			
DIGITAL UNIX			
	Input:	a0 = 4	Function code value
		a1 = arg	Argument from Table E–10
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 4	Function code value
		R17 = arg	Argument from Table E–10
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)
Read the counters			
DIGITAL UNIX			
	Input:	a0 = 5	Function code value
		a1 = arg	Argument from Table E–11
	Output:	v0 = val	Return value from Table E–11
OpenVMS Alpha			
	Input:	R16 = 5	Function code value
		R17 = arg	Argument from Table E–11
	Output:	R0 = val	Return value from Table E–11

Table E–4: OpenVMS Alpha and DIGITAL UNIX Performance Monitoring Functions (Continued)

Function	Register Usage		Comments
Write the counters			
DIGITAL UNIX			
	Input:	a0 = 6	Function code value
		a1 = arg	Argument from Table E–12
	Output:	v0 = 1	Success
		v0 = 0	Failure (not generated)
OpenVMS Alpha			
	Input:	R16 = 6	Function code value
		R17 = arg	Argument from Table E–12
	Output:	R0 = 1	Success
		R0 = 0	Failure (not generated)

Table E–5: 21164/21164PC Enable Counters for OpenVMS Alpha and DIGITAL UNIX

Bits	Meaning When Set
2	Operate on counter 2
1	Operate on counter 1
0	Operate on counter 0

Table E–6: 21164/21164PC Disable Counters for OpenVMS Alpha and DIGITAL UNIX

Bits	Meaning When Set
2	Operate on counter 2
1	Operate on counter 1
0	Operate on counter 0

Table E–7: 21164 Select Desired Events for OpenVMS Alpha and DIGITAL UNIX

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–16.
21:19	CBOX1	CBOX1 event selection (only has meaning when event selection field PCSEL1 is value <15>; otherwise MBZ). CBOX1 described in Table E–15.
18:8		MBZ
7:4	PCSEL1	Counter 1 event selection. PCSEL1 described in Table E–13.
3:0	PCSEL2	Counter 2 event selection. PCSEL2 described in Table E–14.

Table E–8: 21164PC Select Desired Events for OpenVMS Alpha and DIGITAL UNIX

Bits	Name	Meaning
63:32		MBZ
31	PCSEL0	Counter 0 selection: **Value** **Meaning** 0 Cycles 1 Issues
30:14		MBZ
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–18.
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–17.

Table E–8: 21164PC Select Desired Events for OpenVMS Alpha and DIGITAL UNIX (Continued)

Bits	Name	Meaning
7:4	PCSEL1	Counter 1 event selection. PCSEL1 described in Table E–13.
3:0	PCSEL2	Counter 2 event selection. PCSEL2 described in Table E–14.

Table E–9: 21164/21164PC Select Special Options for OpenVMS Alpha and DIGITAL UNIX

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 Alpha, "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:

DIGITAL UNIX counts user mode by using the executive counter; that is, the count for executive mode is returned as the user mode count.

Table E–10: 21164/21164PC Select Desired Frequencies for OpenVMS Alpha and DIGITAL UNIX

Table E–10 contains the selection definitions for each of the three counters. All frequency fields are two-bit fields with the following values defined:

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–11: 21164/21164PC Read Counters for OpenVMS Alpha and DIGITAL UNIX

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–12: 21164/21164PC Write Counters for OpenVMS Alpha and DIGITAL UNIX

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

Table E–13: 21164/21164PC Counter 1 (PCSEL1) Event Selection

The following values choose the 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

Table E–13: 21164/21164PC Counter 1 (PCSEL1) Event Selection (Continued)

The following values choose the counter 1 (PCSEL1) event selection:

Value	Meaning
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–15. For the 21164PC, use PM0_MUX event selection in Table E–17.

Table E–14: 21164/21164PC Counter 2 (PCSEL2) Event Selection

The following values choose the 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–16. For the 21164PC, use PM1_MUX event selection in Table E–18.

Table E–15: 21164 CBOX1 Event Selection

The following values choose the 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

Table E–16: 21164 CBOX2 Event Selection

The following values choose the 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

Table E–17: 21164PC PM0_MUX Event Selection

The following values choose the PM0_MUX event selection and perform the chosen operation in Counter 0.

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

Table E–18: 21164PC PM1_MUX Event Selection

The following values choose the PM1_MUX event selection and perform the chosen operation in Counter 1.

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
5	System read/flush B-cache hits
6	System read/flush B-cache misses
7	Read miss 3 launched

E.2.3 21264 Performance Monitoring

PALcode instructions control the 21264 on-chip performance counters. For OpenVMS Alpha, the instruction is MTPR_PERFMON; for DIGITAL UNIX and Windows NT Alpha, 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 on chip counters count events. Counters can be individually programmed, read, and written.

Processes can be selectively monitored with the PME bit.

Profile monitoring for the 21264 is called aggregate mode profile monitoring because it provides an aggregate count.

E.2.3.1 Performance Monitor Interrupt Mechanism

The performance monitoring interrupt mechanism varies according to the particular operating system.

For the OpenVMS Alpha 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

When the interrupt is taken, the PC is saved on the stack frame as the old PC.

For the DIGITAL UNIX 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 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

For the Windows NT Alpha Operating System

When a counter overflows and interrupt enabling conditions are correct, the counter causes an interrupt to PALcode. The PALcode builds a frame on the kernel stack and dispatches to the kernel at the interrupt entry point.

E.2.3.2 Windows NT Alpha Functions and Argument

The functions for Windows NT Alpha execute on only a single (the current running) processor. The wrperfmon instruction is called with the following input registers:

Input Register	Contents (Bits)	Meaning
a0	63–0	The register in Table E–19, which contains the value to be written to the hardware PCTR_CTL register.
a1	0	When a1 = 0, write a0 to the hardware PCTR_CTL register. When a1 = 1, read the hardware PCTR_CTL register. The returned PCTR_CTL register is written to register v0.

Table E–19: Bit Summary of PCTR_CTL Register for Windows NT Alpha

Bits	Name	Meaning
63–48	SEXT[PCTR0_CTL[47]	
47–28	PCTR0	Counter 0 value. Enabled by setting I_CTL[PCT0_EN] and either I_CTL[SPCE] or PCTX[PPCE]. On overflow, an interrupt is triggered at ISUM[PC0], if enabled by IER_CM[PCEN0]. Mode is determined by SL0 and operation is described in SL1.
27–26	Reserved	
25–6	PCTR1	Counter 1 value. Enabled by setting I_CTL[PCT1_EN] and either I_CTL[SPCE] or PCTX[PPCE]. On overflow, an interrupt is triggered at ISUM[PC1], if enabled by IER_CM[PCEN1]. Operation is described in SL1.
5	Reserved	
4	SL0	PCTR0 input selecter: **Value Meaning** 0 Aggregate counting mode 1 Reserved

Table E–19: Bit Summary of PCTR_CTL Register for Windows NT Alpha

Bits	Name	Meaning	
3–2	SL1	PCTR1 input selector. If SL0 value is 0:	
		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.
1–0	Reserved		

E.2.3.3 OpenVMS Alpha and DIGITAL UNIX Functions and Arguments

The functions execute only on a single (the current running) processor and are described in Table E–20.

The OpenVMS Alpha 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 DIGITAL UNIX 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–20: OpenVMS Alpha and DIGITAL UNIX Performance Monitoring Functions

Function	Register Usage		Comments
Enable performance monitoring			
	DIGITAL UNIX		
	Input:	a0 = 1	Function code value
		a1 = arg	Argument from Table E–21
	OpenVMS Alpha		
	Input:	R16 = 1	Function code value
		R17 = arg	Argument from Table E–21

Table E–20: OpenVMS Alpha and DIGITAL UNIX Performance Monitoring Functions

Function	Register Usage	Comments
Disable performance monitoring		
DIGITAL UNIX		
Input:	a0 = 0	Function code value
	a1 = arg	Argument from Table E–22
OpenVMS Alpha		
Input:	R16 = 0	Function code value
	R17 = arg	Argument from Table E–22
Select desired events (MUX_SELECT)		
DIGITAL UNIX		
Input:	a0 = 2	Function code value
	a1 = arg	Argument from Table E–23
OpenVMS Alpha		
Input:	R16 = 2	Function code value
	R17 = arg	Argument from Table E–23
Select logging options		
DIGITAL UNIX		
Input:	a0 = 3	Function code value
	a1[0] = 1	Log all processes
	a1[0] = 0	Log only selected processes
OpenVMS Alpha		
Input:	R16 = 3	Function code value
	R17[0] = 1	Log all processes
	R17[0] = 0	Log only selected processes
Read the counters		
DIGITAL UNIX		
Input:	a0 = 5	Function code value
Output:	v0 = contents of the counters; see Table E–24	
OpenVMS Alpha		
Input:	R16 = 5	Function code value
Output:	R0 = contents of the counters; see Table E–24	

Table E–20: OpenVMS Alpha and DIGITAL UNIX Performance Monitoring Functions

Function	Register Usage		Comments
Write the counters			
	DIGITAL UNIX		
	Input:	a0 = 6	Function code value
		a1 = arg	Argument from Table E–25
	OpenVMS Alpha		
	Input:	R16 = 6	Function code value
		R17 = arg	Argument from Table E–25
Enable and write selected counters			
	DIGITAL UNIX		
	Input:	a0 = 7	Function code value
		a1 = arg	Argument from Table E–26
	OpenVMS Alpha		
	Input:	R16 = 7	Function code value
		R17 = arg	Argument from Table E–26

Table E–21: 21264 Enable Counters for OpenVMS Alpha and DIGITAL UNIX

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–22: 21264 Disable Counters for OpenVMS Alpha and DIGITAL UNIX

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–23: 21264 Select Desired Events for OpenVMS Alpha and DIGITAL UNIX

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.

Table E–24: 21264 Read Counters for OpenVMS Alpha and DIGITAL UNIX

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–25: 21264 Write Counters for OpenVMS Alpha and DIGITAL UNIX

R17/a1 Bits	Meaning
63–48	Reserved
47–28	Counter 0 value to write
27–26	Reserved
25–6	Counter 1 value to write

Table E–25: 21264 Write Counters for OpenVMS Alpha and DIGITAL UNIX

R17/a1 Bits	Meaning
5–2	Reserved
1	When set, write to Counter 1
0	When set, write to Counter 0

Table E–26: 21264 Enable and Write Counters for OpenVMS Alpha and DIGITAL UNIX

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

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z