Alpha AXP Architecture Handbook

Order Number EC–QD2KA–TE

Revision/Update Information: This is Version 2 of the Alpha AXP Architecture Handbook.

Digital Equipment Corporation
Maynard, Massachusetts

October 1994

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

Contents

Chapter 3 Instruction Formats (I)

Chapter 4 Instruction Descriptions (I)

Chapter 5 System Architecture and Programming Implications (I)

Chapter 6 Common PALcode Architecture (I)

Chapter 7 Console Subsystem Overview (I)

Chapter 8 Input/Output Overview (I)

Chapter 9 OpenVMS AXP

Chapter 10 DEC OSF/1

Chapter 11 Windows NT AXP

Appendix A Software Considerations

Appendix B IEEE Floating-Point Conformance

Appendix C Instruction Summary

Index

Figures

Tables

Preface

Chapters 1 through 8 and appendixes A through C of this book are directly derived from sections of the *Alpha AXP System Reference Manual* and are an accurate representation of the described parts of the Alpha AXP architecture.

Chapter 1
Introduction (I)

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

The Alpha AXP architecture is designed to avoid bias toward any particular operating system or programming language. Alpha AXP supports the OpenVMS AXP, DEC OSF/1, and Windows NT AXP operating systems and supports simple software migration for applications that run on those operating systems.

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

1.1 The Alpha AXP Approach to RISC Architecture

Alpha AXP Is a True 64-Bit Architecture

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

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

Alpha AXP 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 AXP Approach to Byte Manipulation

The Alpha AXP architecture does byte shifting and masking with normal 64-bit register-to-register instructions, crafted to keep instruction sequences short.

Alpha AXP does not include single-byte store instructions. This has several advantages:

- Cache and memory implementations need not include byte shift-and-mask logic, and sequencer logic need not perform read-modify-write on memory locations. Such logic is awkward for high-speed implementation and tends to slow down cache access to normal 32-bit or 64-bit aligned quantities.
- The Alpha AXP approach to byte manipulation makes it easier to build a high-speed error-correcting write-back cache, which is often needed to keep a very fast RISC implementation busy.
- The Alpha AXP approach can make it easier to pipeline multiple byte operations.

The Alpha AXP Approach to Arithmetic Traps

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

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

The Alpha AXP 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 AXP an especially attractive architecture for building multiple-processor systems.

Alpha AXP Instructions Include Hints for Achieving Higher Speed

A number of Alpha AXP 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—The Alpha AXP Very Flexible Privileged Software Library

A Privileged Architecture Library (PALcode) is a set of subroutines that are specific to a particular Alpha AXP 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 AXP implementations run the full OpenVMS AXP, DEC OSF/1, and Windows NT AXP operating systems. PALcode can provide this functionality with little overhead. For example, the OpenVMS AXP PALcode instructions let Alpha AXP run OpenVMS with little more hardware than that found on a conventional RISC machine: the PAL mode bit itself, plus 4 extra protection bits in each translation buffer entry.

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

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

Alpha AXP and Programming Languages

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

- Alpha AXP does not contain a subroutine call instruction that moves a register window by a fixed amount. Thus, Alpha AXP is a good match for programming languages with many parameters and programming languages with no parameters.
- Alpha AXP 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 AXP is a load/store RISC architecture with the following data characteristics:

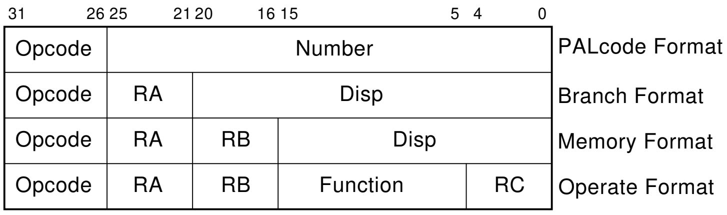
- All operations are done between 64-bit registers.

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

Figure 1–1: Instruction Format Overview



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

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

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

1.4 Instruction Overview

PALcode Instructions

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

Branch Instructions

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

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

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

Load/Store Instructions

Load and store instructions move either 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.

There are facilities for doing byte manipulation in registers, eliminating the need for 8-bit or 16-bit load/store instructions.

Compilers, as directed by user declarations, can generate any mixture of 32-bit and 64-bit operations. The Alpha AXP 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 AXP architecture also supports the following additional operations:

- Scaled add/subtract instructions for quick subscript calculation
- 128-bit multiply for division by a constant, and multiprecision arithmetic
- Conditional move instructions for avoiding branch instructions
- An extensive set of in-register byte and word manipulation instructions

Integer overflow trap enable is encoded in the function field of each instruction, rather than kept in a global state bit. Thus, for example, both ADDQ/V and ADDQ opcodes exist for specifying 64-bit ADD with and without overflow checking. That makes it easier to pipeline implementations.

Floating-Point Operate Instructions

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

In addition to the operations found in conventional RISC architectures, Alpha AXP 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 then kept in global state bits. That makes it easier to pipeline implementations.

1.5 Instruction Set Characteristics

Alpha AXP 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.
- There are 32 floating-point registers (F0 through F31), each 64 bits wide. F31 reads as zero, and writes to F31 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.
- All floating-point data manipulation is between floating-point registers, with up to two register source operands and one register destination operand.
- All memory reference instructions are of the load/store type that move data between registers and memory.
- There are no branch condition codes. Branch instructions test an integer or floating-point register value, which may be the result of a previous compare.
- Integer and logical instructions operate on quadwords.

- Floating-point instructions operate on G_floating, F_floating, IEEE double, and IEEE single operands. D_floating ''format compatibility,'' in which binary files of D_floating numbers may be processed, but without the last 3 bits of fraction precision, is also provided.
- A minimal number of VAX compatibility instructions are included.

1.6 Terminology and Conventions

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

1.6.1 Numbering

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

1.6.2 Security Holes

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

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

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

The Alpha AXP 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 occurences.

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

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

UNPREDICTABLE

- Results or occurrences specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE.
- An UNPREDICTABLE result may acquire an arbitrary value subject to a few constraints. Such a result may be an arbitrary function of the input operands or of any state information that is accessible to the process in its current access mode. UNPREDICTABLE results may be unchanged from their previous values.

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

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

  Also, operations that may produce UNPREDICTABLE results must not:

  – Write or modify the contents of memory locations or registers to which the current process in the current access mode does not have access, or

  – Halt or hang the system or any of its components.

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

UNDEFINED

- Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing, to stopping system operation.
- UNDEFINED operations may halt the processor or cause it to lose information. However, UNDEFINED operations must not cause the processor to hang, that is, reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions.

1.6.4 Ranges and Extents

Ranges are specified by a pair of numbers separated by a ‘‘..’’ and are inclusive. For example, a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.

Extents are specified by a pair of numbers in angle brackets separated by a colon and are inclusive. For example, bits <7:3> specify an extent of bits including bits 7, 6, 5, 4, and 3.

1.6.5 ALIGNED and UNALIGNED

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

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

1.6.6 Must Be Zero (MBZ)

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

1.6.7 Read As Zero (RAZ)

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

1.6.8 Should Be Zero (SBZ)

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

1.6.9 Ignore (IGN)

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

1.6.10 Implementation Dependent (IMP)

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

1.6.11 Figure Drawing Conventions

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

1.6.12 Macro Code Example Conventions

All instructions in macro code examples are listed in Chapter 4 or are stylized code forms found in *Appendix A*.

Chapter 2

Basic Architecture (I)

2.1 Addressing

The basic addressable unit in the Alpha AXP 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.

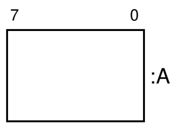
2.2 Data Types

Following are descriptions of the Alpha AXP 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

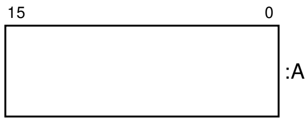


A byte is specified by its address A. A byte is an 8-bit value. The byte is only supported in Alpha AXP by the extract, mask, insert, and zap instructions.

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



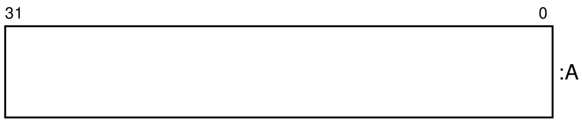
A word is specified by its address, the address of the byte containing bit 0.

A word is a 16-bit value. The word is only supported in Alpha AXP by the 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 AXP by sign-extended load and store instructions and by longword arithmetic instructions.

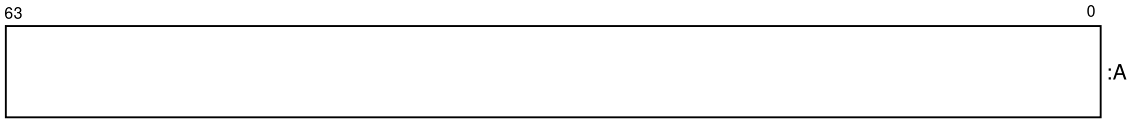
Note:

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

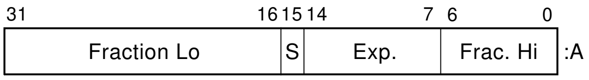
2.2.5 VAX Floating-Point Formats

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

2.2.5.1 F_floating

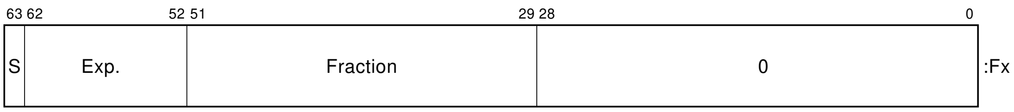
An F_floating datum is 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2–5.

Figure 2–5: F_floating Datum



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

Figure 2–6: F_floating Register Format



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

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

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	

This mapping preserves both normal values and exceptional values.

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

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

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

Note:

> Alpha AXP 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

31 … 16	15	14 … 4	3 … 0	
Fraction Midh	S	Exp.	Frac.Hi	:A
Fraction Lo	Fraction Midl			:A+4

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

Figure 2–8: G_floating Format

63	62 … 52	51 … 32	31 … 0	
S	Exp.	Fraction Hi	Fraction Lo	:Fx

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

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

Note:

> Alpha AXP 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

31 16	15	14 7	6 0	
Fraction Midh	S	Exp.	Frac.Hi	:A
Fraction Lo	Fraction Midl			:A+4

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

63	62 55	54 48	47 32	31 16	15 0	
S	Exp.	Frac. Hi	Fraction Midh	Fraction Midl	Fraction Lo	:Fx

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

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

Notes:

- D_floating is not a fully supported data type; no D_floating arithmetic operations are provided in the architecture. For backward compatibility, exact D_floating arithmetic may be provided via software emulation. D_floating ‘‘format compatibility’’ in which binary files of D_floating numbers may be processed, but without the last 3 bits of fraction precision, can be obtained via conversions to G_floating, G arithmetic operations, then conversion back to D_floating.
- Alpha AXP 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 AXP architecture supports the basic single and double formats, with the basic double format serving as the extended single format. The values representable within a format are specified by using three integer parameters:

1. P—the number of fraction bits
2. Emax—the maximum exponent
3. Emin—the minimum exponent

Within each format, only the following entities are permitted:

1. Numbers of the form (–1)**S x 2**E x b(0).b(1)b(2)..b(P–1) where:
   a. S = 0 or 1
   b. E = any integer between Emin and Emax, inclusive
   c. b(n) = 0 or 1
2. Two infinities—positive and negative
3. At least one Signaling NaN
4. At least one Quiet NaN

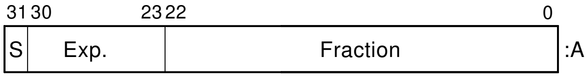
NaN is an acronym for Not-a-Number. A NaN is an IEEE floating-point bit pattern that represents something other than a number. NaNs come in two forms: Signaling NaNs and Quiet NaNs. Signaling NaNs are used to provide values for uninitialized variables and for arithmetic enhancements. Quiet NaNs provide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

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

2.2.6.1 S_Floating

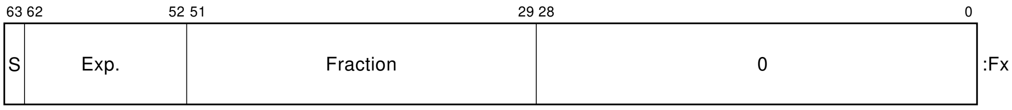
An IEEE single-precision, or S_floating, datum occupies 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2–11.

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:

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

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

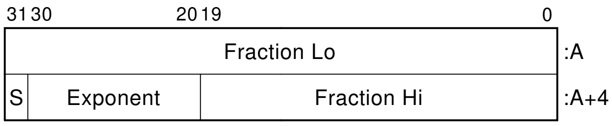
Note:

> Alpha AXP 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

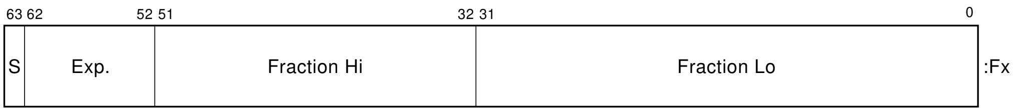
An IEEE double-precision, or T_floating, datum occupies 8 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–13.

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:

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

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

Note:

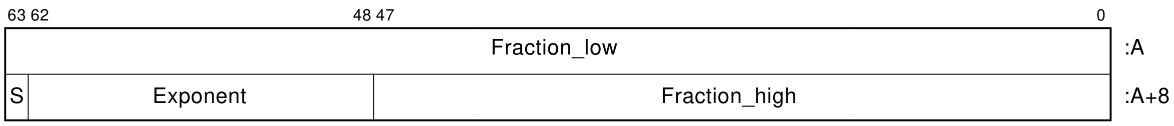
> Alpha AXP implementations will impose a significant performance penalty when accessing T_floating operands that are not naturally aligned. (A naturally aligned T_floating datum has zero as the low-order three bits of its address.)

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.

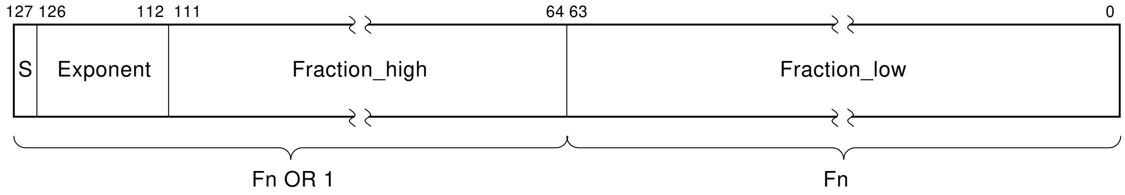
An IEEE extended-precision, or X_floating, datum occupies 16 contiguous bytes in memory, starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 127, as shown in Figure 2–15.

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:

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

Note:

Alpha AXP 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 AXP 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

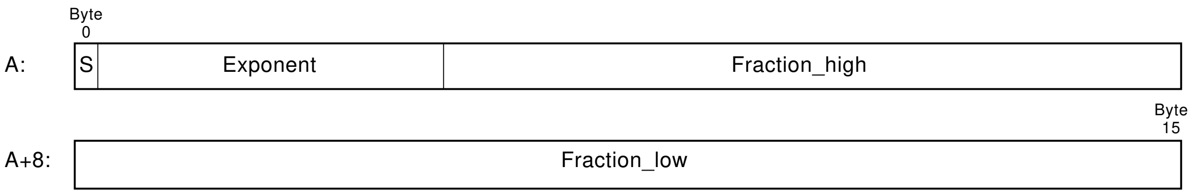
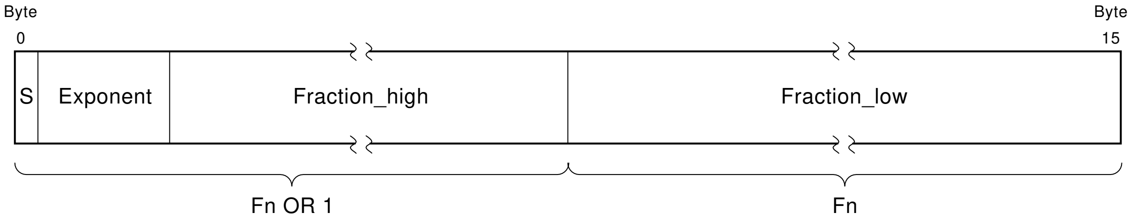


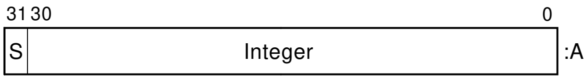
Figure 2–18: X_Floating Big-Endian Register Format



2.2.7 Longword Integer Format in Floating-Point Unit

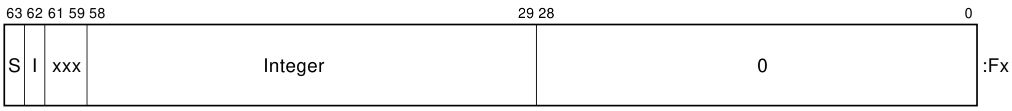
A longword integer operand occupies 32 bits in memory, arranged as shown in Figure 2–19.

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.

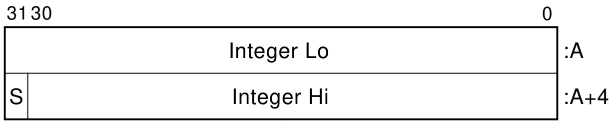
Note:

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

2.2.8 Quadword Integer Format in Floating-Point Unit

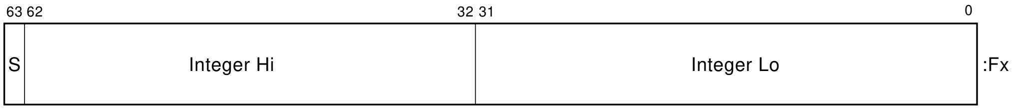
A quadword integer operand occupies 64 bits in memory, arranged as shown in Figure 2–21.

Figure 2–21: Quadword Integer Datum



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

Figure 2–22: Quadword Integer Floating-Register Format



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

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

Note:

> Alpha AXP 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 AXP 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 AXP implementations may include optional big-endian addressing support.

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

Figure 2–23: Little-Endian Byte Addressing

7	6	5	4	3	2	1	0

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

Figure 2–24: Big-Endian Byte Addressing

0	1	2	3	4	5	6	7

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

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

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

- A quadword load or store of data at location 0 moves the same eight bytes under both numbering conventions. However, a longword load or store of data at location 4 must move the leftmost half of a quadword under the little-endian convention, and the rightmost half under the big-endian convention. Thus, to support both conventions, the convention being used must be known and it must affect longword load/store operations.
- A byte extract of byte 5 from a quadword of data into the low byte of a register requires a right shift of 5 bytes under the little-endian convention, but a right shift of 2 bytes under the big-endian convention.
- Manipulating data in a register is almost the same for both conventions. In both, integer and floating-point data have their sign bits in the leftmost byte and their least significant bit in the rightmost byte, so the same integer and

floating-point instructions are used unchanged for both conventions. Big-endian character strings have their most significant character on the left, while little-endian strings have their most significant character on the right.

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

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

- A way 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 then the half which would be accessed under the little-endian convention.
- If the big-endian convention is chosen, the byte manipulation instructions (EXTxx, INSxx, MSKxx) invert bits Rbv<2:0>. This has the effect of changing a shift of 5 bytes into a shift of 2 bytes, for example.

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

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

Chapter 3

Instruction Formats (I)

3.1 Alpha AXP Registers

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

3.1.1 Program Counter

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

3.1.2 Integer Registers

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

Register R31 is assigned special meaning by the Alpha AXP 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. It is also UNPREDICTABLE whether exceptions are signaled during the execution of such an instruction. Note, however, that exceptions associated with the instruction fetch of such an instruction are always signaled.

There are some interesting cases involving R31 as a destination:

- STx_C R31,disp(Rb)

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

- LDx_L R31,disp(Rb)

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

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

3.1.3 Floating-Point Registers

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

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

Results of an instruction that specifies F31 as a destination operand are discarded and it is UNPREDICTABLE whether the other destination operands (implicit and explicit) are changed by the instruction. In this case, it is implementation-dependent to what extent the instruction is actually executed once it has been fetched. It is also UNPREDICTABLE whether exceptions are signaled during the execution of such an instruction. Note, however, that exceptions associated with the instruction fetch of such an instruction are always signaled.

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

3.1.4 Lock Registers

There are two per-processor registers associated with the LDx_L and STx_C instructions, the lock_flag and the locked_physical_address register. The use of these registers is described in Section 4.2.

3.1.5 Processor Cycle Counter (PCC) Register

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

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

PCC_CNT increments once per N CPU cycles, where N is an implementation-specific integer in the range 1..16. The cycle counter frequency is the number of times the processor cycle counter gets incremented per second. The integer count wraps to 0 from a count of FFFF 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 an interrupt occurs before PCC_CNT overflows twice.

PCC_OFF need not contain a value related to time and could contain all zeroes 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.

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 4.11.5.

3.1.6 Optional Registers

Some Alpha AXP 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.

Table 3–2 (Cont.): Operand Value Notation

Notation	Meaning
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>

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

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.

Name	Meaning
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.

<access type>
Is a letter denoting the operand access type:

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

<data type>
Is a letter denoting the data type of the operand:

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 3–4 describes the operators:

Table 3–4: Operators

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>!</td><td>Comment delimiter</td></tr>
<tr><td>+</td><td>Addition</td></tr>
<tr><td>-</td><td>Subtraction</td></tr>
<tr><td>*</td><td>Signed multiplication</td></tr>
<tr><td>*U</td><td>Unsigned multiplication</td></tr>
<tr><td>**</td><td>Exponentiation (left argument raised to right argument)</td></tr>
<tr><td>/</td><td>Division</td></tr>
<tr><td>←</td><td>Replacement</td></tr>
<tr><td>||</td><td>Bit concatenation</td></tr>
<tr><td>{}</td><td>Indicates explicit operator precedence</td></tr>
<tr><td>(x)</td><td>Contents of memory location whose address is x</td></tr>
<tr><td>x<m:n></td><td>Contents of bit field of x defined by bits n through m</td></tr>
<tr><td>x<m></td><td>M’th bit of x</td></tr>
<tr><td>ACCESS(x,y)</td><td>Accessibility of the location whose address is x using the access mode y. Returns a Boolean value TRUE if the address is accessible, else FALSE.</td></tr>
<tr><td>AND</td><td>Logical product</td></tr>
<tr><td>ARITH_RIGHT_SHIFT(x,y)</td><td>Arithmetic right shift of first operand by the second operand. Y is an unsigned shift value. Bit 63, the sign bit, is copied into vacated bit positions and shifted out bits are discarded.</td></tr>
<tr><td>BYTE_ZAP(x,y)</td><td>X is a quadword, y is an 8-bit vector in which each bit corresponds to a byte of the result. The y bit to x byte correspondence is y<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.</td></tr>
</table>

Table 3–4 (Cont.): Operators

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

Table 3–4 (Cont.): Operators

Operator	Meaning
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 set

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

3.2.4 Notation Conventions

The following conventions are used:

1. Only operands that appear on the left side of a replacement operator are modified.
2. No operator precedence is assumed other than that replacement (←) has the lowest precedence. Explicit precedence is indicated by the use of ‘‘{}’’.

3. All arithmetic, logical, and relational operators are defined in the context of their operands. For example, ‘‘+’’ applied to G_floating operands means a G_floating add, whereas ‘‘+’’ applied to quadword operands is an integer add. Similarly, ‘‘LT’’ is a G_floating comparison when applied to G_floating operands and an integer comparison when applied to quadword operands.

3.3 Instruction Formats

There are five basic Alpha AXP 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.

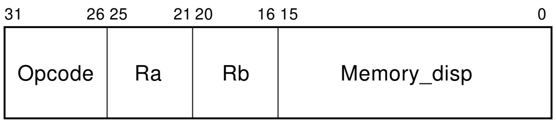
Software Note:

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

31 26	25 21	20 16	15 0
Opcode	Ra	Rb	Function

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

31 26	25 21	20 0
Opcode	Ra	Branch_disp

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

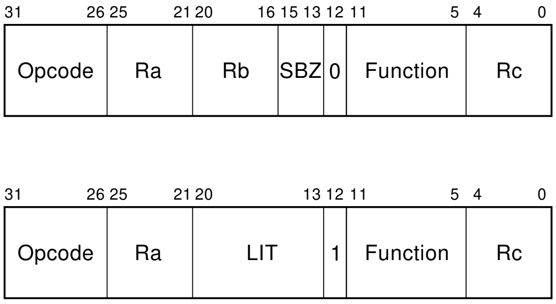
The displacement is treated as a longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits and added to the updated PC to form the target virtual address. Overflow is ignored in this calculation. The target virtual address (va) is computed as follows:

```
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 field. Unused function codes for those opcodes defined as reserved in the Version 5 Alpha AXP architecture specification (May 1992) produce an illegal instruction trap. Those opcodes are 01, 02, 03, 04, 05, 06, 07, 0A, 0C, 0D, 0E, 14, 19, 1B, 1C, 1D, 1E, and 1F. For other opcodes, unused function codes produce UNPREDICTABLE but not UNDEFINED results; they are not security holes.

There are three operand fields, Ra, Rb, and Rc.

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

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

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

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

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

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

The Rc field specifies a destination operand.

3.3.4 Floating-Point Operate Instruction Format

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

Figure 3–5: Floating-Point Operate Instruction Format

31 26	25 21	20 16	15 5	4 0
Opcode	Fa	Fb	Function	Fc

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 AXP architecture specification (May 1992) produce an illegal instruction trap. Those opcodes are 01, 02, 03, 04, 05, 06, 07, 0A, 0C, 0D, 0E, 14, 19, 1B, 1C, 1D, 1E, and 1F. For other opcodes, unused function codes produce UNPREDICTABLE but not UNDEFINED results; they are not security holes.

There are three operand fields, 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.

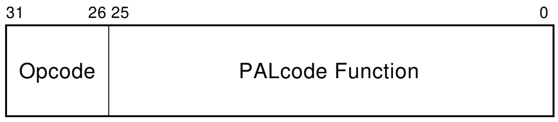
3.3.4.1 Floating-Point Convert Instructions

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.

3.3.5 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 (I)

4.1 Instruction Set Overview

This chapter describes the instructions implemented by the Alpha AXP 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	4.8
Floating-point control	4.9
Floating-point operate	4.10
Miscellaneous	4.11

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 AXP architecture is not performed in either hardware or PALcode. System software may provide emulation routines for subsetted instructions.

4.1.1.1 Floating-Point Subsets

Floating-point support is optional on an Alpha AXP processor. An implementation that supports floating-point must implement the 32 floating-point registers, the Floating-point Control Register (FPCR) and the instructions to access it, floating-point branch instructions, floating-point copy sign (CPYSx) instructions, floating-point convert instructions, floating-point conditional move instruction (FCMOV), and 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 AXP operating system. This requirement facilitates the implementation of a floating-point emulator and simplifies context-switching.

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

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

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

4.1.2 Software Emulation Rules

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

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

4.1.3 Opcode Qualifiers

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

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

Table 4–1: Opcode Qualifiers

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

The default values are normal rounding, software completion disabled, inexact result disabled, floating underflow disabled, and integer overflow disabled.

4.2 Memory Integer Load/Store Instructions

The instructions in this section move data between the integer registers and memory.

They use the Memory instruction format. The instructions are summarized in Table 4–2.

Table 4–2: Memory Integer Load/Store Instructions

Mnemonic	Operation
LDA	Load Address
LDAH	Load Address High
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
STL	Store Longword
STL_C	Store Longword Conditional
STQ	Store Quadword
STQ_C	Store Quadword Conditional
STQ_U	Store Quadword Unaligned

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                                         ! LDL
  big_endian_data:    va' ← va XOR 100₂      ! LDL
  little_endian_data: va' ← va               ! LDL
ENDCASE                                      ! LDL

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

Ra  ← (va)<63:0>                             ! LDQ
```

Exceptions:

Access Violation
Alignment
Fault on Read
Translation Not Valid

Instruction mnemonics:

LDL	Load Sign-Extended Longword from Memory to Register
LDQ	Load Quadword 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 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, and written to register Ra. If the data is not naturally aligned, an alignment exception is generated.

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                                          ! LDL_L
  big_endian_data:    va' ← va XOR 100₂       ! LDL_L
  little_endian_data: va' ← va                ! LDL_L
ENDCASE                                       ! LDL_L

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, the store occurs; otherwise, it does not occur, as described for the STx_C instructions.

If processor A's lock_flag is set and processor B successfully does a store within A's locked range of physical addresses, then A's lock_flag is cleared. 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 quadword) 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's executing a normal load or store instruction. It is UNPREDICTABLE whether a processor's lock_flag is cleared by that processor's executing a taken branch (including BR, BSR, and Jumps); conditional branches that fall through do not clear the lock_flag.

The sequence:

```
LDx_L
Modify
STx_C
BEQ xxx
```

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

Notes:

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

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

  LDx_L and STx_C instructions need not be paired. In particular, an LDx_L may be followed by a conditional branch: on the fall-through path an STx_C is done, whereas on the taken path no matching STx_C is done.

  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 any other memory access (LDx, LDQ_U, STx, STQ_U) is done 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 done 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 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.

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                                          ! STL_C
  big_endian_data:    va’ ← va XOR 100₂       ! STL_C
  little_endian_data: va’ ← va                ! STL_C
ENDCASE                                       ! STL_C

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 the lock_flag is set but the addressing constraints are not met, the outcome of the STx_C instruction is UNPREDICTABLE. In all cases, Ra is set to zero if the STx_C failed (memory was not written) and set to one if it succeeded (memory was written).

If the addressing constraints were not met and the lock_flag was cleared by execution of a CALL_PAL REI, CALL_PAL rti, CALL_PAL rfe, or STx_C instruction since the most recent execution of a LDx_L instruction, a zero is returned in Ra and no write to memory occurs. (See the LDx_L description for conditions that clear the lock_flag.)

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.
- 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 AXP 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 AXP 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.

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                                          ! STL
  big_endian_data:    va' ← va XOR 100₂       ! STL
  little_endian_data: va' ← va                ! STL
ENDCASE                                       ! STL

(va')<31:0> ←  Rav<31:0>                      ! STL
(va) ←  Rav                                   ! STQ
```

Exceptions:

Access Violation
Fault on Write
Alignment
Translation Not Valid

Instruction mnemonics:

STL	Store Longword from Register to Memory
STQ	Store Quadword 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 longword access, va<2> (bit 2 of the virtual address) is 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.

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 AXP 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 AXP architecture includes explicit hints based on a branch-prediction model:

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

The Alpha AXP 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 AXP page and hence are expected to be enough for branch-prediction logic to start an early I-cache access for the most likely target.

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

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

Table 4–3: Control Instructions Summary

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

Notes:

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

4.3.2 Unconditional Branch

Format:

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 *Appendix A* for more information on branch prediction.

An unconditional long jump can be performed by:

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

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

Encoding	Meaning
0000_{16}	Indicates non-procedure return
0001_{16}	Indicates procedure return
	All other encodings are reserved to Digital.

4.4 Integer Arithmetic Instructions

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

The integer instructions are summarized in Table 4–5.

Table 4–5: Integer Arithmetic Instructions Summary

Mnemonic	Operation
ADD	Add Quadword/Longword
S4ADD	Scaled Add by 4
S8ADD	Scaled Add by 8
CMPEQ	Compare Signed Quadword Equal
CMPLT	Compare Signed Quadword Less Than
CMPLE	Compare Signed Quadword Less Than or Equal
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 via UMULH; division by a variable can be done via a subroutine. See *Appendix A*.

4.4.1 Longword Add

Format:

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

Operation:

```
Rc ←  SEXT( (Rav + Rbv)<31:0>)
```

Exceptions:

Integer Overflow

Instruction mnemonics:

ADDL Add Longword

Qualifiers:

Integer Overflow Enable (/V)

Description:

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

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

4.4.2 Scaled Longword Add

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

4.4.3 Quadword Add

Format:

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 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 are written to the destination register.

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

4.4.8 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.

4.4.9 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.

4.4.10 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>.

4.4.11 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.

4.4.12 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), there will be a borrow.

4.4.13 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 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 AXP provides instructions for operating on byte operands within registers. These instructions allow full-width memory accesses in the load/store instructions combined with powerful in-register byte manipulation.

The instructions are summarized in Table 4–7.

Table 4–7: Byte-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
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

Table 4–7 (Cont.): Byte-Manipulation Instructions Summary

Mnemonic	Operation
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/less:

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

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

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

4.6.2 Extract Byte

Format:

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

Operation:

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

CASE

     EXTBL: byte_mask ←  0000 0001₂
     EXTWx: byte_mask ←  0000 0011₂
     EXTLx: byte_mask ←  0000 1111₂
     EXTQx: byte_mask ←  1111 1111₂
ENDCASE

CASE

     EXTxL:

         byte_loc ←  Rbv'<2:0>*8

         temp ←  RIGHT_SHIFT(Rav, byte_loc<5:0>)
         Rc ←  BYTE_ZAP(temp, NOT(byte_mask) )

     EXTxH:

         byte_loc ←  64 - Rbv'<2:0>*8

         temp ←  LEFT_SHIFT(Rav, byte_loc<5:0>)
         Rc ←  BYTE_ZAP(temp, NOT(byte_mask) )
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

EXTBL	Extract Byte Low
EXTWL	Extract Word Low
EXTLL	Extract Longword Low
EXTQL	Extract Quadword Low

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
```

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
```

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
```

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
```

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

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

Optimized examples:

Assume that a word fetch is needed from 10(R3), where R3 is intended to contain a longword-aligned address. The optimized sequences below take advantage of the known constant offset, and the longword alignment (hence a single aligned longword contains the entire word). The sequences generate a Data Alignment Fault if R3 does not contain a longword-aligned address.

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
```

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:

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 111₂
  little_endian_data: Rbv' ← Rbv
ENDCASE

CASE
    INSBL: byte_mask ←  0000 0000 0000 0001₂
    INSWx: byte_mask ←  0000 0000 0000 0011₂
    INSLx: byte_mask ←  0000 0000 0000 1111₂
    INSQx: byte_mask ←  0000 0000 1111 1111₂
ENDCASE

byte_mask ←  LEFT_SHIFT(byte_mask, Rbv'<2:0>)

CASE

    INSxL:

        byte_loc ←  Rbv'<2:0>*8

        temp ←  LEFT_SHIFT(Rav, byte_loc<5:0>)
        Rc ←  BYTE_ZAP(temp, NOT(byte_mask<7:0>))
    INSxH:

        byte_loc ←  64 - Rbv'<2:0>*8

        temp ←  RIGHT_SHIFT(Rav, byte_loc<5:0>)
        Rc ←  BYTE_ZAP(temp, NOT(byte_mask<15:8>))

ENDCASE
```

Exceptions:

None

Instruction mnemonics:

INSBL	Insert Byte Low
INSWL	Insert Word Low
INSLL	Insert Longword Low

INSQL	Insert Quadword Low
INSWH	Insert Word High
INSLH	Insert Longword High
INSQH	Insert Quadword High

Qualifiers:

None

Description:

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

4.6.4 Byte Mask

Format:

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

Operation:

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

CASE
    MSKBL: byte_mask ←  0000 0000 0000 0001₂
    MSKWx: byte_mask ←  0000 0000 0000 0011₂
    MSKLx: byte_mask ←  0000 0000 0000 1111₂
    MSKQx: byte_mask ←  0000 0000 1111 1111₂
ENDCASE

byte_mask ←  LEFT_SHIFT(byte_mask, Rbv'<2:0>)

CASE
    MSKxL:
        Rc ←  BYTE_ZAP(Rav, byte_mask<7:0>)

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

Exceptions:

None

Instruction mnemonics:

MSKBL	Mask Byte Low
MSKWL	Mask Word Low
MSKLL	Mask Longword Low
MSKQL	Mask Quadword Low
MSKWH	Mask Word High
MSKLH	Mask Longword High
MSKQH	Mask Quadword High

Qualifiers:

None

Description:

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

Notes:

The comments in the examples below assume that the effective address (ea) of X(R11) is such that (ea 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 similer 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
```

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
```

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

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

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

Note:

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

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

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

4.7.1 Floating-Point Single-Precision Operations

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

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 Floating Subsets and Floating Faults

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

All floating-point loads and stores may take memory management faults (access control violation, translation not valid, fault on read/write, data alignment).

The floating-point enable (FEN) internal processor register (IPR) allows system software to restrict access to the floating-point registers.

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

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

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

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

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

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

4.7.3 Definitions

The following definitions apply to Alpha AXP floating-point support.

Alpha AXP finite number

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

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

denormal

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

dirty zero

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

infinity

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

LSB
The least significant bit. For a positive representable number A whose fraction is not all ones, A + 1 LSB is the next larger representable number, and A + 1/2 LSB is exactly halfway between A and the next larger representable number.

non-finite number
An IEEE infinity, NaN, or denormal number.

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 AXP, those with an initial fraction bit of 0) and quiet NaNs (for Alpha AXP, 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

Sign	Exponent	Fraction	Vax Meaning	VAX Finite	IEEE Meaning	IEEE Finite
1	0	Non-zero	Resv. operand	No	–Denormal	No
0	0	0	True zero	Yes	+0	Yes
1	0	0	Resv. operand	No	–0	Yes
x	Other	x	Finite	Yes	finite	Yes

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

Data Format	MIN	MAX
F_floating	2**–127 * 0.5 (0.293873588e–38)	2**127 * (1.0 – 2**–24) (1.7014117e38)
G_floating	2**–1023 * 0.5 (0.5562684646268004e–308)	2**1023 * (1.0 – 2**–53) (0.89884656743115785407e308)
S_floating	2**–126 * 1.0 (1.17549435e–38)	2**127 * (2.0 – 2**–23) (3.40282347e38)
T_floating	2**–1022 * 1.0 (2.2250738585072013e–308)	2**1023 * (2.0 – 2**–52) (1.7976931348623158e308)
X_floating	2**–16382 * 1.0 (3.362103143112093506262677817321752600e–4932)	2**16383 * (2.0 – 2**–112) (1.18973149535723176508575932662800702e4932)

4.7.5 Floating-Point Rounding Modes

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

VAX Rounding Modes

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

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

Chopped VAX rounding maps the true result to the smaller in magnitude of two surrounding representable results; maps true results ≥ 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 4.7.7). Alpha AXP 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 4.7.7.

The following tables summarize the floating-point rounding modes:

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

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

4.7.6 Floating-Point Trapping Modes

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

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

VAX Trapping Modes

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

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

IEEE Trapping Modes

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

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

The modes and instruction notation are:

VAX Trap Mode	Instruction Notation
Imprecise, underflow disabled	(No modifier)
Imprecise, underflow enabled	/U
Software, underflow disabled	/S
Software, underflow enabled	/SU

VAX Convert-to-Integer Trap Mode	Instruction Notation
Imprecise, integer overflow disabled	(No modifier)
Imprecise, integer overflow enabled	/V
Software, integer overflow disabled	/S

VAX Convert-to-Integer Trap Mode	Instruction Notation
Software, integer overflow enabled	/SV

IEEE Trap Mode	Instruction Notation
Imprecise, unfl disabled, inexact disabled	(No modifier)
Imprecise, unfl enabled, inexact disabled	/U
Software, unfl enabled, inexact disabled	/SU
Software, unfl enabled, inexact enabled	/SUI

IEEE Convert-to-Integer Trap Mode	Instruction Notation
Imprecise, int.ovfl disabled, inexact disabled	(No modifier)
Imprecise, int.ovfl enabled, inexact disabled	/V
Software, int.ovfl enabled, inexact disabled	/SV
Software, int.ovfl enabled, inexact enabled	/SVI

4.7.6.1 Imprecise /Software Completion Trap Modes

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

- The trapping instruction may write an UNPREDICTABLE result value.
- The trap PC is an arbitrary number of instructions past the one triggering the trap. The trigger instruction plus all intervening executed instructions are collectively referred to as the *trap shadow* of the trigger instruction.
- The extent of the trap shadow is bounded only by a TRAPB instruction (or the implicit TRAPB within a CALL_PAL instruction).
- Input operand values may have been overwritten in the trap shadow.
- Result values may have been overwritten in the trap shadow.
- An UNPREDICTABLE result value may have been used as an input operand in the trap shadow.
- Additional traps may occur in the trap shadow.
- In general, it is not feasible to fix up the result value or to continue from the trap.

This behavior is ideal for operations on finite operands that give finite results. For programs that deliberately operate outside the overflow/underflow range, or use IEEE NaNs, software assistance is required to complete floating-point operations correctly. This assistance can be provided by a software arithmetic trap handler, plus constraints on the instructions surrounding the trap.

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

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

1. If the value in a register or memory location is used as input to some instruction in the trap shadow, then either the following condition *a* or condition *b* must be met.

   a. The register or memory location is not modified by the instruction that uses it or by any subsequent instruction in the trap shadow.

   b. The value was produced by an earlier instruction in the trap shadow, and no trapping instruction appears between the producing and consuming instructions.

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

2. If a conditional move (CMOVxx or FCMOVxx) instruction appears in the trap shadow, then the Ra/Fa and Rb/Fb operands of the instruction must satisfy condition 1 above and either the following condition *a* or condition *b* must be met.

   a. The Ra/Fa operand of the conditional move does not depend on any value produced earlier in the trap shadow by an instruction that might trap.

   b. The Rc/Fc operand of the instruction was produced by an earlier instruction in the trap shadow, and no trapping instruction appears between the producing instruction and the conditional move.

   Condition *a* ensures that the conditional move instruction will be reexecuted with the same condition in Ra/Fa. If condition *a* cannot be ensured, then condition *b* requires that the value that might be overlaid is valid when the conditional move is reexecuted.

3. If a value is produced in the trap shadow as the result of a floating-point instruction that might trap, that value may not contribute to any value that is subsequently used in the trap shadow as the input to an integer instruction that has the /V modifier.

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

5. Within the trap shadow, if a register is used as the destination register of an instruction that might cause a floating-point exception (and thus set a bit in the software implementation's exception summary), it may not be used as the destination of any other instruction in the trap shadow.
6. The trap shadow may not include any branch instructions.
7. Each floating-point instruction to be completed must be so marked, by specifying the /S software completion modifier. The /S modifier must not be used on any floating-point instruction that is not in a trap shadow that meets these conditions.

Note:

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

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

4.7.6.2 Invalid Operation (INV) Arithmetic Trap

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

- Any operation on a signaling NaN.
- Addition of unlike-signed infinities or subtraction of like-signed infinities, such as (+infinity $+$ –infinity) or (+infinity $-$ +infinity).
- Multiplication of 0 $*$infinity.
- 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.

An implementation may also choose to signal an invalid operation when it encounters an operand that is non-finite. However, CMPTxy does not trap on plus or minus infinity.

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

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

4.7.6.3 Division by Zero (DZE) Arithmetic Trap

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

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

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

4.7.6.4 Overflow (OVF) Arithmetic Trap

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

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

4.7.6.5 Underflow (UNF) Arithmetic Trap

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

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

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

4.7.6.6 Inexact Result (INE) Arithmetic Trap

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

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

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

4.7.6.7 Integer Overflow (IOV) Arithmetic Trap

In conversions from floating to quadword integer, an integer overflow occurs if the rounded result is outside the range –2**63..2**63–1. In conversions from quadword integer to longword integer, an integer overflow occurs if the result is outside the range –2**31..2**31–1.

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

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

4.7.6.8 Floating-Point Trap Disable Bits

Any of the traps described in Sections 4.7.6.2 through 4.7.6.7 may be disabled by setting the corresponding trap disable bit in the FPCR.

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. If the implementation is unable to calculate the required result, it ignores the trap disable bit and signals a trap as usual. (When an implementation supports both the underflow disable bit and the underflow to zero bit, and both bits are set in the FPCR, the implementation sets the result of an underflow operation to an appropriately signed zero value.)

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

4.7.7 FPCR Register and Dynamic Rounding Mode

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

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

Floating-point operates that belong to the IEEE subset and CVTQL, which belongs to both VAX and IEEE subsets, appropriately set the FPCR exception bits. It is UNPREDICTABLE whether floating-point operates that belong only to the VAX floating-point subset set the FPCR exception bits.

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

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

- If the UNFD bit is not implemented, the hardware may not implement the UNDZ bit.

- If the DZED bit is implemented, division of 0/0 must be treated as an invalid operation instead of a division by zero.

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

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

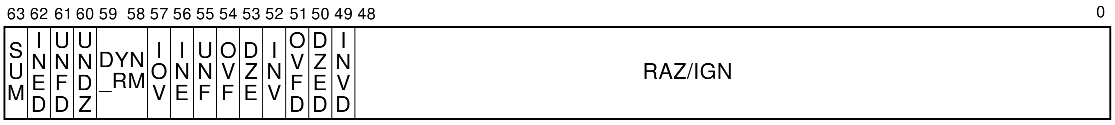


Table 4–8: Floating-Point Control Register (FPCR) Bit Descriptions

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 if the implementation is capable of producing correct IEEE nontrapping result.
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 an appropriately signed zero in the destination register rather than the denormal number 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

Bit	Description (Meaning When Set)
57	Integer Overflow (IOV). An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision.

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

Bit	Description (Meaning When Set)
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 result.
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 result.
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 result.
48–0	Reserved. Read As Zero; Ignored when written.

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

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

Software Note:

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

4.7.7.1 Accessing the FPCR

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

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

Consider the following example:

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

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

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

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

4.7.7.2 Default Values of the FPCR

Processor initialization leaves the value of FPCR UNPREDICTABLE.

Software Note:

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

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

4.7.7.3 Saving and Restoring the FPCR

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

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

Software Notes:

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

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

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

   > Trap interface software (usually the operating system) keeps a set of software status bits and a mask of the traps that the user wants to receive. Code is generated with the /SUI modifiers. For a particular exception, the software sets the corresponding trap disable bit if either the corresponding software status bit is 0 or if the user wants to receive such traps. If a trap occurs, the software locates the offending instruction in the trap shadow, simulates it and sets any of the software status bits that are appropriate. Then, the software either delivers the trap to the user program or disables further delivery of such traps. The user program must interface to this trap interface software to set or clear any of the software status bits or to enable or disable floating-point traps.

   When such a scheme is being used, the trap disable bits should be modified only by the trap interface software. If the disable bits are spuriously cleared, unnecessary traps may occur. If they are spuriously set, the software may fail to set the correct values in the software status bits. Programs should call routines in the trap interface software to set or clear bits in the FPCR.

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

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

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

4.7.8 Floating-Point Computational Models

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

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

IEEE Compliant Arithmetic

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

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

IEEE Compliant Arithmetic Without Inexact Exception

This is similar to the previous model, but it does not provide the inexact exception status bit, nor does it allow a program to request traps when an inexact operation occurs.

This model is implemented in a program by using IEEE floating-point operates with the /SU modifiers, with the help of the trap interface software.

High-Performance IEEE-Format Arithmetic

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

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

4.7.9 Floating-Point Instruction Function Field Format

Bits <15..5> in floating-point instructions contain the function field, as shown in Figure 4–2 and described for IEEE floating-point in Table 4–9 and for VAX floating-point in Table 4–10. 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

31–26	25–21	20–16	15–13	12–11	10–9	8–5	4–0
Opcode	Fa	Fb	TRP	RND	SRC	FNC	Fc

Table 4–9: IEEE Floating-Point Function Field Bit Summary

Bits	Field	Meaning
15–13	TRP	Trapping modes: 000 Imprecise (default) 001 Underflow enable (/U) — floating-point output Integer overflow enable (/V) — integer output 010 Unsupported 011 Unsupported 100 Software completion enable (/S) 101 /SU — floating-point output /SV — integer output 110 Unsupported 111 /SUI — floating-point output /SVI — integer output
12–11	RND	Rounding modes: 00 Chopped (/C) 01 Minus infinity (/M) 10 Normal (default) 11 Dynamic (/D)
10–9	SRC	Source datatype: 00 S_floating 01 Reserved 10 T_floating 11 Q_fixed

Table 4–9 (Cont.): IEEE Floating-Point Function Field Bit Summary

Bits	Field	Meaning
8–5	FNC	Instruction class:
		0000 ADDx
		0001 SUBx
		0010 MULx
		0011 DIVx
		0100 CMPxUN
		0101 CMPxEQ
		0110 CMPxLT
		0111 CMPxLE
		1000 Reserved
		1001 Reserved
		1010 Reserved
		1011 Reserved
		1100 CVTxS
		1101 Reserved
		1110 CVTxT
		1111 CVTxQ

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

Bits	Field	Meaning
15–13	TRP	Trapping modes:
		000 Imprecise (default)
		001 Underflow enable (/U) — floating-point output Integer overflow enable (/V) — integer output
		010 Unsupported
		011 Unsupported
		100 Software completion enable (/S)
		101 /SU — floating-point output /SV — integer output
		110 Unsupported
		111 Unsupported
12–11	RND	Rounding modes:
		00 Chopped (/C)
		01 Unsupported
		10 Normal (default)
		11 Unsupported

Table 4–10 (Cont.): VAX Floating-Point Function Field Bit Summary

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

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 AXP architecture in these situations. Note that this behavior may be supplied by either hardware (if the invalid operation disable, or INVD, bit is implemented) or by software. See Sections 4.7.6.8, 4.7.7, and 4.7.7.3.

4.7.10.1 Conversion of NaN and Infinity Values

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

Conversion of a NaN value from S_floating to T_floating gives a result identical to the input, except that the most significant fraction bit (bit 51) is set to indicate a quiet NaN.

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

4.7.10.2 Copying NaN Values

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 AXP architecture, the result of such an operation is a NaN generated according to the first of the following rules that is applicable:

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

4.8 Memory Format Floating-Point Instructions

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

The instructions are summarized in Table 4–11.

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

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 100₂
  little_endian_data: va' ← va
ENDCASE

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

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDS Load S_floating (Load Longword Integer)

Qualifiers:

None

Description:

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

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

4.8.4 Load T_floating

Format:

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

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

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

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

Table 4–12: Floating-Point Branch Instructions Summary

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, and floating convert operations on 64-bit register values in one of the four specified floating formats.

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

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

Table 4–13: Floating-Point Operate Instructions Summary

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 4–13 (Cont.): Floating-Point Operate Instructions Summary

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
MULF	Multiply F_floating	VAX
MULG	Multiply G_floating	VAX
MULS	Multiply S_floating	IEEE
MULT	Multiply T_floating	IEEE

Table 4–13 (Cont.): Floating-Point Operate Instructions Summary

Mnemonic	Operation	Subset
Arithmetic Operations		
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:	Software (/S) (CVTQL only)
	Integer Overflow Enable (/V) (CVTQL only)

Description:

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

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

The conversion from longword to quadword is a repositioning of 32 bits of the operand, with sign extension.

4.10.3 Floating-Point Conditional Move

Format:

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

Operation:

```
IF  TEST(Fav, Condition_based_on_Opcode)  THEN
    Fc ←  Fbv
```

Exceptions:

None

Instruction mnemonics:

FCMOVEQ	FCMOVE if Register Equal to Zero
FCMOVGE	FCMOVE if Register Greater Than or Equal to Zero
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
   MT_FPCR:  FPCR ←  Fav
   MF_FPCR:  Fa   ←  FPCR
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 4.7.7.

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:	Software (/S)
	Underflow Enable (/U)

Description:

Register Fa is added to register Fb, and the sum is written to register Fc.

The sum is rounded or chopped to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs. See Section 4.7.6 for details of the stored result on overflow or underflow.

4.10.6 IEEE Floating Add

Format:

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:	Software (/S)
	Underflow Enable (/U)
	Inexact Enable (/I)

Description:

Register Fa is added to register Fb, and the sum is written to register Fc.

The sum is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.

4.10.7 VAX Floating Compare

Format:

CMPGyy Fa.rg,Fb.rg,Fc.wq !Floating-point Operate format

Operation:

```
IF  Fav SIGNED_RELATION Fbv  THEN
    Fc ←  4000 0000 0000 0000₁₆
ELSE
    Fc ←  0000 0000 0000 0000₁₆
```

Exceptions:

Invalid Operation

Instruction mnemonics:

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

Qualifiers:

Trapping:	Software (/S)

Description:

The two operands in Fa and Fb are compared. If the relationship specified by the qualifier is true, a non-zero floating value (0.5) is written to register Fc; otherwise, a true zero is written to Fc.

Comparisons are exact and never overflow or underflow. Three mutually exclusive relations are possible: less than, equal, and greater than.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

Notes:

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

4.10.8 IEEE Floating Compare

Format:

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

Operation:

```
IF  Fav SIGNED_RELATION Fbv  THEN
     Fc ←  4000 0000 0000 0000₁₆
ELSE
    Fc ←  0000 0000 0000 0000₁₆
```

Exceptions:

Invalid Operation

Instruction mnemonics:

CMPTEQ	Compare T_floating Equal
CMPTLE	Compare T_floating Less Than or Equal
CMPTLT	Compare T_floating Less Than
CMPTUN	Compare T_floating Unordered

Qualifiers:

Trapping:	Software (/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 (2.0) is written to register Fc; otherwise, a true zero is written to Fc.

Comparisons are exact and never overflow or underflow. Four mutually exclusive relations are possible: less than, equal, greater than, and unordered. The unordered relation is true if one or both operands are NaN. (This behavior must be provided by a software trap handler, since NaNs trap.) Comparisons ignore the sign of zero, so +0 = –0.

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

Notes:

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

4.10.9 Convert VAX Floating to Integer

Format:

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

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

Invalid Operation
Integer Overflow

Instruction mnemonics:

CVTGQ	Convert G_floating to Quadword

Qualifiers:

Rounding:	Chopped (/C)
Trapping:	Software (/S)
	Integer Overflow Enable (/V)

Description:

The floating operand in register Fb is converted to a two's-complement quadword number and written to register Fc. The conversion aligns the operand fraction with the binary point just to the right of bit zero, rounds as specified, and complements the result if negative. Register Fa must be F31.

An invalid operation trap is signaled if the operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.6 for details of the stored result on integer overflow.

4.10.10 Convert Integer to VAX Floating

Format:

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:	Software (/S)
	Underflow Enable (/U)

Description:

The floating operand in register Fb is converted to the specified alternate floating format and written to register Fc. Register Fa must be F31.

An invalid operation trap is signaled if the operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.6 for details of the stored result on overflow or underflow.

Notes:

- The only arithmetic operations on D_floating values are conversions to and from G_floating. The conversion to G_floating rounds or chops as specified, removing three fraction bits. The conversion from G_floating to D_floating adds three low-order zeros as fraction bits, then the 8-bit exponent range is checked for overflow /underflow.
- The conversion from G_floating to F_floating rounds or chops to single precision, then the 8-bit exponent range is checked for overflow/underflow.
- No conversion from F_floating to G_floating is required, since F_floating values are always stored in registers as equivalent G_floating values.

4.10.12 Convert IEEE Floating to Integer

Format:

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:	Software (/S)
	Integer Overflow Enable (/V)
	Inexact Enable (/I)

Description:

The floating operand in register Fb is converted to a two's-complement number and written to register Fc. The conversion aligns the operand fraction with the binary point just to the right of bit zero, rounds as specified, and complements the result if negative. Register Fa must be F31.

See Section 4.7.6 for details of the stored result on integer overflow and inexact result.

4.10.13 Convert Integer to IEEE Floating

Format:

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:	Software (/S)
	Inexact Enable (/I)

Description:

The two's-complement operand in register Fb is converted to a single- or double-precision floating result and written to register Fc. The conversion complements a number if negative, normalizes it, rounds to the target precision, and packs the result with an appropriate sign and exponent field. Register Fa must be F31.

See Section 4.7.6 for details of the stored result on inexact result.

4.10.14 Convert IEEE S_Floating to IEEE T_Floating

Format:

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

Operation:

```
Fc <- {conversion of Fbv}
```

Exceptions:

Invalid Operation

Instruction mnemonics:

CVTST Convert S_floating to T_floating

Qualifiers:

Trapping: Software (/S)

Description:

The S_floating operand in register Fb is converted to T_floating format and written to register Fc. Register Fa must be F31.

Notes:

- The conversion from S_floating to T_floating is exact. No rounding occurs. No underflow, overflow, or inexact result can occur. In fact, the conversion for finite values is the identity transformation.
- A trap handler can convert an S_floating denormal value into the corresponding T_floating finite value by adding 896 to the exponent and normalizing.

4.10.15 Convert IEEE T_Floating to IEEE S_Floating

Format:

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

Operation:

```
Fc ←  {conversion of 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:	Software (/S)
	Underflow Enable (/U)
	Inexact Enable (/I)

Description:

The T_floating operand in register Fb is converted to S_floating format and written to register Fc. Register Fa must be F31.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.

4.10.16 VAX Floating Divide

Format:

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:	Software (/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 4.7.6 for details of the stored result on overflow or underflow.

4.10.17 IEEE Floating Divide

Format:

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

Operation:

```
Fc ←  Fav / Fbv
```

Exceptions:

Invalid Operation
Division by Zero
Overflow
Underflow
Inexact Result

Instruction mnemonics:

DIVS	Divide S_floating
DIVT	Divide T_floating

Qualifiers:

Rounding:	Dynamic (/D)
	Minus infinity (/M)
	Chopped (/C)
Trapping:	Software (/S)
	Underflow Enable (/U)
	Inexact Enable (/I)

Description:

The dividend operand in register Fa is divided by the divisor operand in register Fb, and the quotient is written to register Fc.

The quotient is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.

4.10.18 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:	Software (/S)
	Underflow Enable (/U)

Description:

The multiplicand operand in register Fb is multiplied by the multiplier operand in register Fa, and the product is written to register Fc.

The product is rounded or chopped to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.6 for details of the stored result on overflow or underflow.

4.10.19 IEEE Floating Multiply

Format:

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

Operation:

```
Fc ←  Fav * Fbv
```

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

MULS	Multiply S_floating
MULT	Multiply T_floating

Qualifiers:

Rounding:	Dynamic (/D)
	Minus infinity (/M)
	Chopped (/C)
Trapping:	Software (/S)
	Underflow Enable (/U)
	Inexact Enable (/I)

Description:

The multiplicand operand in register Fb is multiplied by the multiplier operand in register Fa, and the product is written to register Fc.

The product is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.

4.10.20 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:	Software (/S)
	Underflow Enable (/U)

Description:

The subtrahend operand in register Fb is subtracted from the minuend operand in register Fa, and the difference is written to register Fc.

The difference is rounded or chopped to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.6 for details of the stored result on overflow or underflow.

4.10.21 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:	Software (/S)
	Underflow Enable (/U)
	Inexact Enable (/I)

Description:

The subtrahend operand in register Fb is subtracted from the minuend operand in register Fa, and the difference is written to register Fc.

The difference is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.

4.11 Miscellaneous Instructions

Alpha AXP provides the miscellaneous instructions shown in Table 4–14.

Table 4–14: Miscellaneous Instructions Summary

Mnemonic	Operation
CALL_PAL	Call Privileged Architecture Library Routine
EXCB	Exception Barrier
FETCH	Prefetch Data
FETCH_M	Prefetch Data, Modify Intent
MB	Memory Barrier
RPCC	Read Processor Cycle Counter
TRAPB	Trap Barrier
WMB	Write Memory Barrier

4.11.1 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.

4.11.2 Exception Barrier

Format:

EXCB ! Memory format

Operation:

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

Exceptions:

None

Instruction mnemonics:

EXCB Exception Barrier

Qualifiers:

None

Description:

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

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

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

The relationship between EXCB and the FPCR is described in Section 4.7.7.1.

4.11.3 Prefetch Data

Format:

FETCHx 0(Rb.ab) !Memory format

Operation:

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

Exceptions:

None

Instruction mnemonics:

FETCH	Prefetch Data
FETCH_M	Prefetch Data, Modify Intent

Qualifiers:

None

Description:

The virtual address is given by Rbv. This address is used to designate an aligned 512-byte block of data. An implementation may optionally attempt to move all or part of this block (or a larger surrounding block) of data to a faster-access part of the memory hierarchy, in anticipation of subsequent Load or Store instructions that access that data.

The FETCH instruction is a hint to the implementation that may allow faster execution. An implementation is free to ignore the hint. If prefetching is done in an implementation, the order of fetch within the designated block is UNPREDICTABLE.

The FETCH_M instruction gives the additional hint that modifications (stores) to some or all of the data block are anticipated.

No exceptions are generated by FETCHx. If a Load (or Store in the case of FETCH_M) that uses the same address would fault, the prefetch request is ignored. It is UNPREDICTABLE whether a TB-miss fault is ever taken by FETCHx.

Implementation Note:

Implementations are encouraged to take the TB-miss fault, then continue the prefetch.

Software Note:

FETCH is intended to help software overlap memory latencies on the order of 100 cycles. FETCH is unlikely to help (or be implemented) for memory latencies on the order of 10 cycles. Code scheduling should be used to overlap such short latencies.

The programming model for effective use of FETCH and FETCH_M is given in *Appendix A*.

4.11.4 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.

4.11.5 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 to not cause an unwanted sign extension.

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

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

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

4.11.6 Trap Barrier

Format:

TRAPB !Memory format

Operation:

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

Exceptions:

None

Instruction mnemonics:

TRAPB Trap Barrier

Qualifiers:

None

Description:

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

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

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

Use of TRAPB should be compared with use of the EXCB instruction; see Section 4.11.2.

4.11.7 Write Memory Barrier

Format:

WMB !Memory format

Operation:

```
{Guarantee that all preceding stores have accessed
memory before any subsequent stores access memory}
```

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 will not be aggregated with writes that follow the WMB. It also guarantees that all writes preceding the WMB instruction are completed before any writes that follow the WMB instruction. The WMB instruction effectively causes writes contained in buffers to be completed without unnecessary delay. It is particularly suited for batching writes to high-performance I/O devices.

In the absence of a WMB instruction, stores to memory or non-memory-like regions can be aggregated and/or buffered and completed in any order.

The WMB instruction provides for high-bandwidth write streams where order must be maintained between certain writes in that stream. It is the preferred means for programs to obtain this result.

4.12 VAX Compatibility Instructions

Alpha AXP provides the instructions shown in Table 4–15 for use in translated VAX code. These instructions are not a permanent part of the architecture and will not be available in some future implementations. They are intended to preserve customer assumptions about VAX instruction atomicity in porting code from VAX to Alpha AXP.

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

Table 4–15: 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 AXP instructions between RS and RC (corresponding to a single VAX instruction) was executed without interruption or exception.

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

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

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

Note:

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

Chapter 5

System Architecture and Programming Implications (I)

5.1 Introduction

Portions of the Alpha AXP 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 AXP architecture, software and hardware implementors need to take these issues into consideration.

5.2 Physical Address Space Characteristics

Alpha AXP physical address space is divided into four equal-size regions. The regions are delineated by the two most significant, implemented, physical address bits. Each regions 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 AXP implementations must provide a coherent view of memory, in which each write by a processor or I/O device (hereafter, called ''processor'') becomes visible to all other processors. No distinction is made between coherency of ''memory space'' and ''I/O space''.

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

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

1. No caching

No copies are kept of data in a region; all reads and writes access the actual data location (memory or I/O register), but a processor may elide multiple accesses to the same data (see Section 5.2.3).

2. Write-through caching

   Copies are kept of any data in the region; reads may use the copies, but writes update the actual data location and either update or invalidate all copies.

3. Write-back caching

   Copies are kept of any data in the region; reads and writes may use the copies, and writes use additional state to determine whether there are other copies to invalidate or update.

Software/Hardware Note:

To produce separate and distinct accesses to a specific location, the location must be a region with no caching and a memory barrier instruction must be inserted between accesses. See Section 5.2.3.

Part of the coherency policy implemented for a given physical address region may include restrictions on excess data transfers (performing more accesses to a location than is necessary to acquire or change the location's value), or may specify data transfer widths (the granularity used to access a location).

Independent of coherency policy, a processor may use different hardware or different hardware resource policies for caching or buffering different physical address regions.

5.2.2 Granularity of Memory Access

For each region, an implementation must support aligned quadword access and may optionally support aligned longword access.

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.

In this context, ‘‘atomic’’ means that if different processors do simultaneous reads and writes of the same data, it must not be possible to observe a partial write of the subject longword or quadword.

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 obeys 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.
- 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.
- 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 AXP 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 a virtual instruction cache (virtual I-cache) or a virtual data cache (virtual D-cache). A system may also choose to include either a combined data and instruction translation buffer (TB) or separate data and instruction TBs (DTB and ITB). The contents of these caches and/or translation buffers may become invalid, depending on what operating system activity is being performed.

Whenever a non-software field of a valid page table entry (PTE) is modified, copies of that PTE must be made coherent. PALcode mechanisms are available to clear all TBs, both DTB and ITB entries for a given VA, either DTB or ITB entries for a given VA, or all entries with the address space match (ASM) bit clear. Virtual D-cache entries are made coherent whenever the corresponding DTB entry is requested to be cleared by any of the appropriate PALcode mechanisms. Virtual I-cache entries can be made coherent via the CALL_PALL IMB instruction.

If a processor implements address space numbers (ASNs), and the old PTE has the Address Space Match (ASM) bit clear (ASNs in use) and the Valid bit set, then entries can also effectively be made coherent by assigning a new, unused ASN to the currently running process and not reusing the previous ASN before calling the appropriate PALcode routine to invalidate the translation buffer (TB).

In a multiprocessor environment, making the TBs and/or caches coherent on only one processor is not always sufficient. An operating system must arrange to perform the above actions on each processor that could possibly have copies of the PTE or data for any affected page.

5.4 Caches and Write Buffers

A hardware implementation may include mechanisms to reduce memory access time by making local copies of recently used memory contents (or those expected to be used) or by buffering writes to complete at a later time. Caches and write buffers are examples of these mechanisms. They must be implemented so that their existence is transparent to software (except for timing, error reporting/control/recovery, and modification to the I-stream).

The following requirements must be met by all cache/write-buffer implementations. All processors must provide a coherent view of memory.

1. Write buffers may be used to delay and aggregate writes. From the viewpoint of another processor, buffered writes appear not to have happened yet. (Write buffers must not delay writes indefinitely. See Section 5.6.1.10.)

2. Write-back caches must be able to detect a later write from another processor and invalidate or update the cache contents.

3. A processor must guarantee that a data store to a location followed by a data load from the same location must read the updated value.

4. Cache prefetching is allowed, but virtual caches must not prefetch from invalid pages.

5. A processor must guarantee that all of its previous writes are visible to all other processors before a HALT instruction completes. A processor must guarantee that its caches are coherent with the rest of the system before continuing from a HALT.

6. If battery backup is supplied, a processor must guarantee that the memory system remains coherent across a powerfail/recovery sequence. Data that was written by the processor before the powerfail may not be lost, and any caches must be in a valid state before (and if) normal instruction processing is continued after power is restored.

7. Virtual instruction caches are not required to notice modifications of the virtual I-stream (they need not be coherent with the rest of memory). Software that creates or modifies the instruction stream must execute a CALL_PAL IMB before trying to execute the new instructions.

   For example, if two different virtual addresses, VA1 and VA2, map to the same page frame, a store to VA1 modifies the virtual I-stream fetched via VA2.

   However, the following sequence does not modify the virtual I-stream (this might happen in soft page faults).

   1. Change the mapping of an I-stream page from valid to invalid.
   2. Copy the corresponding page frame to a new page frame.
   3. Change the original mapping to be valid and point to the new page frame.

8. Physical instruction caches are not required to notice modifications of the physical I-stream (they need not be coherent with the rest of memory), except for certain paging activity. (See Section 5.6.1.10.) 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.

In this context, to ‘‘modify the virtual I-stream’’ means any Store to the same physical address that is subsequently fetched as an instruction via some corresponding (virtual address, ASN) pair, or to change the virtual-to-physical address mapping so that different values are fetched.

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 via the branches to no_store and try_again. This repetition continues until the reasons for exceptions or interrupts are removed, and no interfering store is encountered.

To be useful, the sequence must be constructed so that it can be replayed an arbitrary number of times, giving the same result values each time. A sufficient (but not necessary) condition is that, within the sequence, the set of operand destinations and the set of operand sources are disjoint.

Note:

> A sufficiently long instruction sequence between LDQ_L and STQ_C will never complete, because periodic timer interrupts will always occur before the sequence completes. The rules in *Appendix A* describe sequences that will eventually complete in *all* Alpha AXP 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_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>

        WMB or MB

        STQ   R31,lock_variable     ; Clear lock bit
         :
         :
already_set:
        <code to block or reschedule or test for too many iterations>
        BR    spin_loop
stq_c_fail:
        <code to test for too many iterations>
        BR    stq_c_loop
```

This code has a number of subtleties:

1. If the lock_variable is already set, the spin loop is done without doing any stores. This avoidance of stores improves memory subsystem performance and avoids the deadlock described below.
2. If the lock_variable is actually being changed from 0 to 1, and the STQ_C fails (due to an interrupt, or because another processor simultaneously changed lock_ variable), the entire process starts over by reading the lock_variable again.
3. Only the fall-through path of the BLBS does a STx_C; some implementations may not allow a successful STx_C after a branch-taken.
4. Only register-to-register operate instructions are used to do the modify.
5. Both conditional branches are forward branches, so they are properly predicted not to be taken (to match the common case of no contention for the lock).
6. The OR writes its result to a second register; this allows the OR and the BLBS to be interchanged if that would give a faster instruction schedule.
7. Other operate instructions (from the critical section) may be scheduled into the LDQ_L..STQ_C sequence, so long as they do not fault or trap, and they give correct results if repeated; other memory or operate instructions may be scheduled between the STQ_C and BEQ.

8. The memory barrier instructions are discussed in Section 5.5.4.

9. An ordinary STQ instruction is used to clear the lock_variable.

It would be a performance mistake to spin-wait by repeating the full LDQ_L..STQ_C sequence (to move the BLBS after the BEQ) because that sequence may repeatedly change the software lock_variable from ‘‘locked’’ to ‘‘locked,’’ with each write causing extra access delays in all other caches that contain the lock_variable. In the extreme, spin-waits that contain writes may deadlock as follows:

> If, when one processor spins with writes, another processor is modifying (not changing) the lock_variable, then the writes on the first processor may cause the STx_C of the modify on the second processor always to fail.

This deadlock situation is avoided by:

- Having only one processor execute a store (no STx_C), or
- Having no write in the spin loop, or
- Doing a write *only* if the shared variable actually changes state (1 → 1 does not change state).

5.5.4 Ordering Considerations for Shared Data Structures

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

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

In the absence of explicit instructions to the contrary, the Alpha AXP 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>

WMB or MB (memory barrier #2)

<clear software lock>
```

The first memory barrier prevents any reads (from within the critical section) from being prefetched before the software lock is acquired; such prefetched reads would potentially contain stale data.

The second memory barrier prevents any writes (and reads if MB is used instead of WMB) from within the critical section from 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.

Hardware/Software Note:

If a WMB is used as the second memory barrier, the programmer is probably relying on causal ordering. Causal ordering is established by cause and effect; the cause occurs before the effect in time order. A causal ordering is contained in the sequence of reading a location or set of locations, performing a calculation on that data, writing the result of that calculation, and then executing a WMB. See Section 5.6.1.2.

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 AXP architecture, programmers must pay more attention to synchronizing access to shared data; for example, to AST routines. In the VAX, a programmer can use an ADDL2 to update a variable that is shared between a ‘‘MAIN’’ routine and an AST routine, if running on a single processor. In the Alpha AXP architecture, a programmer must deal with AST shared data by using multiprocessor shared data sequences.

5.6 Read/Write Ordering

This section does not apply to programs that run on a single processor and do not write to the instruction stream. On a single processor, all memory accesses appear to happen in the order specified by the programmer. This section deals entirely with predictable read/write ordering across multiple processors.

The order of reads and writes done in an Alpha AXP 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 AXP 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 AXP 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 AXP 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 AXP 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 (CALL_PAL IMB, MB or WMB) between the accesses.

5.6.1 Alpha AXP Shared Memory Model

An Alpha AXP 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 AXP 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 an aligned quadword, 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(x,a).
2. Data read by processor *i* to location *x*, returning value *a*, denoted Pi:R(x,a).
3. Data write by processor *i* to location *x*, storing value *a*, denoted Pi:W(x,a).
4. Memory barrier instruction issued by processor *i*, denoted Pi:MB.
5. Write memory barrier instruction issued by processor *i*, denoted Pi:WMB.
6. I-stream memory barrier instruction 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 collectively are called read/write accesses, denoted Pi:*(x,a). The last three types collectively are called barriers or memory barriers.

Instruction fetches are longword reads. Data reads and data writes are either aligned longword or aligned quadword accesses. Unless otherwise noted, it is assumed that each access to a given location has the same access size (that is, if a location is written as a longword it is read as a longword). Section 5.6.1.5 describes the effect of access size on the Alpha AXP shared memory model.

All accesses in this chapter are naturally aligned accesses.

During actual execution in an Alpha AXP system, each processor has a time-ordered issue sequence of all the memory accesses presented by that processor (to all memory locations), and each location has a time-ordered access sequence of all the accesses presented to that location (from all processors).

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 AXP 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 Processor Issue Order

A partial ordering, called processor issue order, is imposed on the issue sequence defined in Section 5.6.1.1.

For two accesses *u* and *v* issued by processor *Pi*, *u* is said to PRECEDE *v* IN ISSUE ORDER (<) if *u* occurs earlier than *v* in the issue sequence for *Pi*, and either of the following applies:

1. The access types are of the following issue order:

Table 5–1: Processor Issue Order

1st ↓ 2nd →	Pi:I(y,b)	Pi:R(y,b)	Pi:W(y,b)	Pi:MB	Pi:WMB	Pi:IMB
Pi:I(x,a)	< if x=y		< if x=y	<	<	<
Pi:R(x,a)		< if x=y	< if x=y	<		<
Pi:W(x,a)		< if x=y	< if x=y	<	<	<
Pi:MB		<	<	<	<	<
Pi:WMB			<	<	<	<
Pi:IMB	<	<	<	<	<	<

2. Or, *u* is a TB fill, 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 Section 5.6.2).

Causal ordering is established by cause and effect; the cause occurs before the effect in time order. A causal ordering is contained in the processor issue sequence of reading a set of locations, performing a calculation on the data fetched by those reads, writing the result of that calculation, and then executing a WMB. The reads that produced the results used in the calculation precede (in issue order) the WMB.

Hardware/Software Note:

The issue order created by causality does not imply that a read precedes (in issue order) a write because of causality. The read is ordered with respect to a subsequent WMB. Also, issue order is not guaranteed in cases in which the result of a calculation on the read data is always independent of the actual value read.

For example, in the following sequence, the LDQ does not necessarily precede the WMB in issue order:

```
LDQ R1,0(R10)
XOR R1,R1,R1
STQ R1,0(R11)
WMB
```

But in the following sequence, both LDQ instructions precede the WMB in issue order even if one of the LDQ intructions reads zero:

```
LDQ R1,0(R10)
LDQ R2,104(R10)
AND R1,R2,R2
STQ R2,0(R11)
WMB
```

Issue order is thus a partial order imposed on the architecturally specified issue sequence. Implementations are free to perform memory accesses from a single processor in any sequence that is consistent with this partial order.

Note that accesses to different locations are ordered only with respect to barriers and TB fill. The table asymmetry for I-fetch allows writes to the I-stream to be incoherent until a CALL_PAL IMB is executed.

5.6.1.3 Definition of Memory Access Sequence

The access sequence for a location 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 AXP implementations.

5.6.1.4 Definition of Location Access Order

A partial ordering, called location access order, is imposed on the memory access sequence defined above.

As shown in Table 5–2, for two accesses *u* and *v* to location *x*, *u* is said to PRECEDE *v* IN ACCESS ORDER ($\ll$) if *u* occurs earlier than *v* in the access sequence for *x*, and at least one of them is a write. Also note in Table 5–2 that processor *Pi* might or might not be the same processor as *Pj*.

Table 5–2: Location Access Order

1st ↓ 2nd →	Pj:I(x,b)	Pj:R(x,b)	Pj:W(x,b)
Pi:I(x,a)			$\ll$
Pi:R(x,a)			$\ll$
Pi:W(x,a)	$\ll$	$\ll$	$\ll$

Access order is thus a partial order imposed on the actual access sequence for a given location. Each location has a separate access order. There is no direct ordering relationship between accesses to different locations.

Note that reads and I-fetches are ordered only with respect to writes.

5.6.1.5 Effect of Access Size

Typically, all accesses to a given location are the same size; a given location is accessed by longword or quadword accesses. Additional issue order and access order rules are imposed when accesses to a given location are not all the same size. The rules differ between aligned quadword read accesses and all other aligned accesses.

The model access for all aligned accesses except an aligned quadword read access is an indivisible event in issue and access order.

The model access for an aligned quadword read access is as two distinct aligned longword read accesses — one to the low-order part of the location and one to the high-order part. Those two longword read accesses are not necessarily identically ordered with respect to other accesses. However, the two model longword read accesses can be identically ordered with respect to other accesses and are then said to occur at the same time in issue order and access order as follows:

- Two accesses, *x* and *y*, are said to occur at the same time in access order if for every access *z*:

  if z $\ll$ x then z $\ll$ y and
  if z $\ll$ y then z $\ll$ x and
  if x $\ll$ z then y $\ll$ z and

if y $\ll$ z then x $\ll$ z.

- Similarly, two accesses, *x* and *y*, are said to occur at the same time in issue order if for every access *z*:

  if z < x then z < y and
  if z < y then z < x and
  if x < z then y < z and
  if y < z then x < z.

For model accesses on a given processor, an aligned longword access has an issue order with respect to a given aligned quadword access if the longword access is to a longword within the quadword.

Table 5–3: Processor Issue Order With Access Size Effect

1st ↓ 2nd →	Pi:I(y,b)	Pi:Rlong(y,b)	Pi:Wlong(y,b)	Pi:Wquad(y,b)	Pi:MB	Pi:WMB	Pi:IMB
Pi:I(x,a)	< if x=y		< if x=y	< if x=y or x=y+4	<	<	<
Pi:Rlong(x,a)		< if x=y	< if x=y	< if x=y or x=y+4	<		<
Pi:Wlong(x,a)		< if x=y	< if x=y	< if x=y or x=y+4	<	<	<
Pi:Wquad(x,a)		< if x=y or x+4=y	< if x=y or x+4=y	< if x=y	<	<	<
Pi:MB		<	<	<	<	<	<
Pi:WMB			<	<	<	<	<
Pi:IMB	<	<	<	<	<	<	<

For model accesses, aligned longword accesses have an access order with respect to a given aligned quadword write access if the access is to a longword within the quadword. Of course, two model accesses have an access order if both have the same access size, both access the same location, and at least one is a write access. Note that in Table 5–4, processor *Pi* might or might not be the same processor as *Pj*.

Table 5–4: Location Access Order With Access Size Effect

1st ↓ 2nd →	Pj:I(x,b)	Pj:Rlong(x,b)	Pj:Wlong(x,b)	Pj:Wquad(x,b)
Pi:I(x,a)			$\ll$ if x=y	$\ll$ if x=y
Pi:Rlong(x,a)			$\ll$ if x=y	$\ll$ if x=y or x=y+4
Pi:Wlong(x,a)	$\ll$ if x=y	$\ll$ if x=y	$\ll$ if x=y	$\ll$ if x=y or x=y+4
Pi:Wquad(x,a)	$\ll$ if x=y or x=y+4	$\ll$ if x=y or x=y+4	$\ll$ if x=y or x+4=y	$\ll$ if x=y

If either of the following conditions is true, the two model longword read accesses that represent a given aligned quadword read access are constrained to occur at the same time in access order and issue order. That is, for the model accesses Pi:Rlong(x,a0) and Pi:Rlong(x+4,a1) modeling aligned quadword access Pi:Rquad(x,a), if either of the following conditions is true, any access that is ordered with respect to Pi:Rlong(x,a0) is identically ordered with respect to Pi:Rlong(x+4,a1).

1. There is an MB or a quadword write access to location *x* by processor *i* after (in processor issue sequence) any longword write access by processor *i* (Pi:Wlong(x or x+4,b)) and before the model read accesses (Pi:Rlong(x,a0) and Pi:Rlong(x+4,a1)).
2. There is no preceding (in processor issue sequence) longword write access to the quadword location by processor *i* (Pi:Wlong(x or x+4,b)), which is later in access order than a write by another processor (processor *j*) that precedes (in access order) either of the model read accesses.

   That is, for every

   Pj:W(x,c) $\ll$ Pi:Rlong(x,a0)

   or

   Pj:W(x or x+4,c) $\ll$ Pi:Rlong(x+4,a1),

   the following holds:

   Pi:Wlong(x or x+4,b) $\ll$ Pj:W(x or x+4,c).

Otherwise, the two model longword accesses might not be identically ordered.

5.6.1.6 Definition of Storage

If *u* is Pi:W(x,a), and *v* is either Pj:I(x,b) or Pj:R(x,b), and u$\ll$v, and no *w* Pk:W(x,c) exists such that u$\ll$w$\ll$v, then the value *b* returned by *v* is exactly the value *a* written by *u*.

Conversely, if *u* is Pi:W(x,a), and *v* is either Pj:I(x,b) or Pj:R(x,b), and b=a (and *a* is distinguishable from values written by accesses other than *u*), then u$\ll$v and for any other *w* Pk:W(x,c) either w$\ll$u or v$\ll$w.

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 processor Pj is modeled as Pi writing to a location INTij, and Pj reading from INTij.)

5.6.1.7 Relationship Between Issue Order and Access Order

If *u* is Pi:*(x,a), and *v* is Pi:*(x,b), one of which is a write, and u<v in the issue order for processor Pi, then u$\ll$v in the access order for location *x*.

In other words, if two accesses to the same location are ordered on a given processor, they are ordered in the same way at the location.

5.6.1.8 Definition of Before

For two accesses *u* and *v*, *u* is said to be BEFORE v ($\Leftarrow$) if:

u < v or
u $\ll$ v or

there exists an access w such that:

(u < w and w $\Leftarrow$ v) or
(u $\ll$ w and w $\Leftarrow$ v).

In other words, ‘‘before’’ is the transitive closure over issue order and access order.

5.6.1.9 Definition of After

If u $\Leftarrow$ v, then *v* is said to be AFTER *u*.

At most one of u $\Leftarrow$ v and v $\Leftarrow$ u is true.

5.6.1.10 Timeliness

Even in the absence of a barrier after the write, a write by one processor to a given location may not be delayed indefinitely in the access order for that location.

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 applying the ordering rules described previously to demonstrate whether the accesses in question are ordered at all.

Assume, in the litmus tests below, that initially all memory locations contain 1.

5.6.2.1 Litmus Test 1 (Impossible Sequence)

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

V1 reading 2 implies U1 $\ll$ V1, by the definition of storage
V2 reading 1 implies V2 $\ll$ U1, by the definition of storage
V1 < V2, by the definition of issue order

The first two orderings imply that V2 $\Leftarrow$ V1, whereas the last implies that V1 $\Leftarrow$ V2.

Both implications cannot be true. 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)

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

V2 reading 2 implies V1 $\Leftarrow$ U1
V3 reading 3 implies U1 $\Leftarrow$ V1

Both implications cannot be true. Thus, once a processor 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)

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

U2 reading 3 implies U1 $\Leftarrow$ V1
W2 reading 2 implies V1 $\Leftarrow$ U1

Both implications cannot be true. Again, time cannot go backward. If U2 reads 3 then W2 must read 3. Alternately, if W2 reads 2, then U2 must read 2.

5.6.2.4 Litmus Test 4 (Sequence Okay)

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

There are no conflicts in this sequence. U2 $\Leftarrow$ V1 and V2 $\Leftarrow$ U1. U1 and U2 are not ordered with respect to each other. V1 and V2 are not ordered with respect to each other. There is no conflicting implication that U1 $\Leftarrow$ V2.

5.6.2.5 Litmus Test 5 (Sequence Okay)

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

There are no conflicts in this sequence. U2 $\Leftarrow$ V1 $\Leftarrow$ V3 $\Leftarrow$ U1. There is no conflicting implication that U1 $\Leftarrow$ U2.

5.6.2.6 Litmus Test 6 (Sequence Okay)

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

There are no conflicts in this sequence. V2 $\Leftarrow$ U1 $\Leftarrow$ U3 $\Leftarrow$ V1. There is no conflicting implication that V1 $\Leftarrow$ V2.

In scenarios 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)

Pi	**Pj**
[U1] Pi:W(x,2)	[V1] Pj:R(y,2)
[U2] Pi:MB or Pi:WMB	[V2] Pj:MB
[U3] Pi:W(y,2)	[V3] Pj:R(x,1)

V1 reading 2 implies U3 ⇐ V1
V3 reading 1 implies V3 ⇐ U1
But, by transitivity, U1 ⇐ U3 ⇐ V1 ⇐ V3

Both cannot be true, so if V1 reads 2, then V3 must also read 2.

5.6.2.8 Litmus Test 8 (Impossible Sequence)

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

U3 reading 1 implies U3 ⇐ V1
V3 reading 1 implies V3 ⇐ U1
But, by transitivity, U1 ⇐ U3 ⇐ V1 ⇐ V3

Both 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)

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

V3 reading 2 implies U1 ⇐ V3
V2 ⇐ V3 and V2 reading 3 implies V2 ⇐ U1
V1 ⇐ V2 and V2 ⇐ U1 implies V1 ⇐ U1

U3 reading 3 implies V1 ⇐ U3
U2 ⇐ U3 and U2 reading 2 implies U2 ⇐ V1
U1 ⇐ U2 and U2 ⇐ V1 implies U1 ⇐ V1

Both V1 $\Leftarrow$ U1 and U1 $\Leftarrow$ V1 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:Wquad(x,200000002_{16})	[V1] Pj:Wlong(x+4,3)
[U2] Pi:Rlong(x+4,3)	[V2] Pj:Rquad(x,300000001_{16})

Model that case as follows:

Pi	Pj
[U1] Pi:Wquad(x,200000002_{16})	[V1] Pj:Wlong(x+4,3)
[U2] Pi:Rlong(x+4,3)	[V2’] Pj:Rlong(x,1)
	[V2’’] Pj:Rlong(x+4,3)

V2’’ returning 3 implies U1 $\ll$ V1 < V2’’. That, in turn, implies that V2’ and V2’’ are not constrained to occur at the same time in access order or issue order. The result of U2 implies U1 $\ll$ V1 $\ll$ U2, whereas the result of V2’ implies V2’ $\ll$ U1. There are no inconsistencies because V2’ and V2’’ can be in different order.

5.6.2.11 Litmus Test 11 (Impossible Sequence)

For an aligned quadword location, *x*, initially 100000001_{16}:

Pi	Pj
[U1] Pi:Wlong(x,2)	[V1] Pj:MB
[U2] Pi:WMB or Pi:MB	[V2] Pj:Rquad(x,200000001_{16})
[U3] Pi:Wlong(x+4,2)	

Model that case as follows:

Pi	Pj
[U1] Pi:Wlong(x,2)	[V1] Pj:MB
[U2] Pi:WMB or Pi:MB	[V2’] Pj:Rlong(x,1)
[U3] Pi:Wlong(x+4,2)	[V2’’] Pj:Rlong(x+4,2)

With no longword write access to *x* between V1 and V2’ or V2’’, V2’ and V2’’ are constrained to occur at the same time in access and issue order. V2’ reading 1 implies V2’ $\ll$ U1, whereas V2’’ reading 2 implies U3 $\ll$ V2’’. This leads to

V2' ≪ U1 < U2 < U3 ≪ V2'', which violates the constraint that V2' and V2'' occur at the same time. The sequence is impossible.

5.6.3 Implied Barriers

In Alpha AXP, there are no implied barriers. 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 AXP 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 Multiple-Processor Data Stream (Including Single Processor with DMA I/O)

Generally, the only way to reliably communicate shared data is to write the shared data on one processor or DMA I/O device, execute an MB or WMB (or the logical equivalent if it is a DMA I/O device), then write a flag (equivalently, send an interrupt) signaling the other processor that the shared data is ready. Each receiving processor must read the new flag (equivalently, receive the interrupt), execute an MB, then read or update the shared data. In the special case in which data is communicated through just one location in memory, memory barriers are not necessary.

Software Note:

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

Leaving out the first MB or WMB 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 before posting a completion interrupt, and the interrupt handler software must execute an MB before the data is guaranteed to be visible to the interrupted processor. Other processors must also execute MBs before they are guaranteed to see the new data.

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

5.6.4.4 Multiple-Processor Instruction Stream (Including Single Processor with DMA I/O)

The only way to update the I-stream reliably is to write the shared I-stream on one processor or DMA I/O device, then execute a CALL_PAL IMB (or an MB or WMB 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), then 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 (MB or WMB) removes the assurance that the shared I-stream is written before the flag.

Leaving out the second CALL_PAL IMB removes the assurance that the shared I-stream is read only *after* the flag is seen to change; in this case, an early read could see an old value.

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

An important special case occurs under the following circumstances:

1. A write (perhaps by an I/O device) is done to some physical page frame.
2. A CALL_PAL IMB (or MB) is executed.
3. A previously invalid PTE is changed to be a valid mapping of the physical page frame that was written in step 1.

In this case, all processors that access virtual memory by using the newly valid PTE must guarantee to deliver the newly written I-stream after the TB miss.

5.6.4.5 Multiple-Processor Context Switch

If a process migrates from executing on one processor to executing on another, the context switch operating system code must include a number of barriers.

A process migrates by having its context stored into memory, then eventually having that context reloaded on another processor. In between, some shared mechanism must be used to communicate that the context saved in memory by the first processor is available to the second processor. This could be done by using an interrupt, by using a flag bit associated with the saved context, or by using a shared-memory multiprocessor data structure, as follows:

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

MB or WMB [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:

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.
		:

First Processor	Second Processor
	Execute code for new process that accesses memory that is not common to all processes.

Note that on a single processor there is no need for the barriers.

5.6.4.6 Multiple-Processor Send/Receive Interrupt

If one processor writes some shared data, then sends an interrupt to a second processor, and that processor receives the interrupt, then accesses the shared data, the sequence from Section 5.6.4.3 must be used:

First Processor		Second Processor
:		
Write data		
MB or WMB		
Send interrupt	⇒	Receive interrupt
		MB
		Access data
		:

Leaving out the MB or WMB 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 AXP 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 AXP 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 or WMB 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.
3. Read the shared data.

As shown in Section 5.6.4.3, leaving out the memory barrier removes the assurance that the shared data is written before the flag is. Unlike the case in Section 5.6.4.3, writing the shared data to a non-memory-like physical memory region removes the assurance that the I/O device will detect the writes of the shared data before detecting the flag write, interrupt, or device register write.

This implies that after a processor has prepared a data buffer to be read from memory by a DMA I/O device (such as writing a buffer to disk), the processor must execute an MB or WMB 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 AXP 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 AXP processor wrote those locations).
2. If an Alpha AXP 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.

5.6.5 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, WMB, 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).

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 AXP implementations are allowed to execute multiple instructions concurrently and to forward results from one instruction to another. Thus, when an arithmetic trap is detected, the PC may have advanced an arbitrarily large number of instructions past the instruction T (calculating result R) whose execution triggered the trap.

When the trap is detected, any or all of these subsequent instructions may run to completion before the trap is actually taken. Instruction T and the set of instructions subsequent to T that complete before the trap is taken are collectively called the trap shadow of T. The PC pushed on the stack when the trap is taken is the PC of the first instruction past the trap shadow.

The instructions in the trap shadow of T may use the UNPREDICTABLE result R of T, they may generate additional traps, and they may completely change the PC (branches, JSR).

Thus, by the time a trap is taken, the PC pushed on the stack may bear no useful relationship to the PC of the trigger instruction T, and the state visible to the programmer may have been updated using the UNPREDICTABLE result R. If an instruction in the trap shadow of T uses R to calculate a subsequent register value, that register value is UNPREDICTABLE, even though there may be no trap associated with the subsequent calculation. Similarly:

- If an instruction in the trap shadow of T stores R or any subsequent UNPREDICTABLE result, the stored value is UNPREDICTABLE.
- If an instruction in the trap shadow of T uses R or any subsequent UNPREDICTABLE result as the basis of a conditional or calculated branch, the branch target is UNPREDICTABLE.
- If an instruction in the trap shadow of T uses R or any subsequent UNPREDICTABLE result as the basis of an address calculation, the memory address actually accessed is UNPREDICTABLE.

Software that is intended to bound how far the PC may advance before taking a trap, or how far an UNPREDICTABLE result may propagate, must insert TRAPB instructions at appropriate points.

Software that is intended to continue from a trap by supplying a well-defined result R within an arithmetic trap handler, can do so reliably by following the rules for software completion code sequences given in Section 4.7.6.

Chapter 6

Common PALcode Architecture (I)

6.1 PALcode

In a family of machines, both users and operating system implementors require functions to be implemented consistently. When functions conform to a common interface, the code that uses those functions can be used on several different implementations without modification.

These functions range from the binary encoding of the instruction and data to the exception mechanisms and synchronization primitives. Some of these functions can be implemented cost effectively in hardware, but others are impractical to implement directly in hardware. These functions include low-level hardware support functions such as Translation Buffer miss fill routines, interrupt acknowledge, and vector dispatch. They also include support for privileged and atomic operations that require long instruction sequences.

In the VAX, these functions are generally provided by microcode. This is not seen as a problem because the VAX architecture lends itself to a microcoded implementation.

One of the goals of Alpha AXP 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 AXP 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 AXP 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 AXP Privileged Architecture Library (PALcode) of routines and environments is supplied by Digital. Other systems may use a library supplied by Digital or architect and implement a different library of routines. Alpha AXP systems are required to support the replacement of PALcode defined by Digital with an operating system-specific version.

6.3 PALcode Environment

The PALcode environment differs from the normal environment in the following ways:

- Complete control of the machine state.
- Interrupts are disabled.
- Implementation-specific hardware functions are enabled, as described below.
- I-stream memory management traps are prevented (by disabling I-stream mapping, mapping PALcode with a permanent TB entry, or by other mechanisms).

Complete control of the machine state allows all functions of the machine to be controlled. Disabling interrupts allows the system to provide multi-instruction sequences as atomic operations. Enabling implementation-specific hardware functions allows access to low-level system hardware. Preventing I-stream memory management traps allows PALcode to implement memory management functions such as translation buffer fill.

6.4 Special Functions Required for PALcode

PALcode uses the Alpha AXP 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 AXP implementation may also choose to provide additional functions to simplify or improve performance of some PALcode functions. The following are some examples:

- An Alpha AXP 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 AXP loads and stores continue to operate on virtual address in the PALcode environment.
- An Alpha AXP 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 AXP 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 AXP systems are required to support the replacement of PALcode supplied by Digital with an operating system-specific version. The following functions must be implemented in PALcode, *not* directly in hardware, to facilitate replacement with different versions.

1. Translation Buffer fill. Different operating systems will want to replace the Translation Buffer (TB) fill routines. The replacement routines will use different data structures. Page tables will not be present in these systems. Therefore, no portion of the TB fill flow that would change with a change in page tables may be placed in hardware, unless it is placed in a manner that can be overridden by PALcode.

2. Process structure. Different operating systems might want to replace the process context switch routines. The replacement routines will use different data structures. The HWPCB or PCB will not be present in these systems. Therefore, no portion of the context switching flows that would change with a change in process structure may be placed in hardware.

PALcode can be viewed as consisting of the following somewhat intertwined components:

- Chip/architecture component
- Hardware platform component
- Operating system component

PALcode should be written modularly to facilitate the easy replacement or conditional building of each component. Such a practice simplifies the integration of CPU hardware, system platform hardware, console firmware, operating system software, and compilers.

PALcode subsections that are commonly subject to modification include:

- Translation Buffer fill
- Process structure and context switch
- Interrupt and exception frame format and routine dispatch
- Privileged PALcode instructions
- Transitions to and from console I/O mode
- Power-up reset

6.7 Required PALcode Instructions

The PALcode instructions listed in Table 6–1 and *Appendix C* must be recognized by mnemonic and opcode in all operating system implementations, but the effect of each instruction is dependent on the implementation. Digital defines the operation of these PALcode instructions for operating system implementations supplied by Digital.

Table 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

Table 6–1 (Cont.): PALcode Instructions that Require Recognition

Mnemonic	Name
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 AXP 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<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<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.

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 Chapter 5.

Chapter 7
Console Subsystem Overview (I)

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

1. Initializes, tests, and prepares the system platform hardware for Alpha AXP system software.
2. Bootstraps (loads into memory and starts the execution of) system software.
3. Controls and monitors the state and state transitions of each processor in a multiprocessor system.
4. Provides services to system software that simplify system software control of and access to platform hardware.
5. Provides a means for a *console operator* to monitor and control the system.

The console 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 interacts with system software once control of the system platform hardware has been transferred to that software.

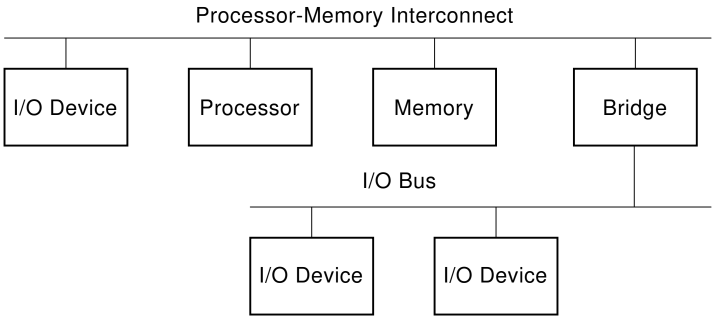
The console interacts with the console operator through a virtual display device or *console terminal*. The console operator may be a human being or a management application.

Chapter 8
Input/Output Overview (I)

Conceptually, Alpha AXP 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 AXP system overview.

Figure 8–1: Alpha AXP 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 AXP 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 AXP

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

9.1 Unprivileged OpenVMS AXP PALcode Instructions

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 AXP PALcode instructions.

Table 9–1: Unprivileged OpenVMS AXP 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.
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.
CHMK	Change mode to kernel The CHMK instruction allows a process to change its mode to kernel in a controlled manner. A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PS, and PC are pushed onto the kernel stack. The saved PC addresses the instruction following the CHMK instruction.

Table 9–1 (Cont.): Unprivileged OpenVMS AXP PALcode Instruction Summary

Mnemonic	Operation and Description
CHMS	Change mode to supervisor The CHMS 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. R2..R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHMS instruction.
CHMU	Change mode to user The CHMU instruction 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.
GENTRAP	Generate trap The GENTRAP instruction is provided for reporting runtime software conditions. It switches the processor to kernel mode and pushes registers R2..R7, the updated PC, and the PS on the kernel stack. It then dispatches to the address of the GENTRAP vector, stored in a control block.
IMB	I-Stream memory barrier The IMB instruction ensures that the contents of an instruction cache are coherent after the instruction stream has been modified by software or I/O devices. 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.

Table 9–1 (Cont.): Unprivileged OpenVMS AXP PALcode Instruction Summary

Mnemonic	Operation and Description
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.
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.

Table 9–1 (Cont.): Unprivileged OpenVMS AXP PALcode Instruction Summary

Mnemonic	Operation and Description
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. PROBE is only intended to check a single datum for accessibility.
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.

Table 9–1 (Cont.): Unprivileged OpenVMS AXP PALcode Instruction Summary

Mnemonic	Operation and Description
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.
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.

Table 9–1 (Cont.): Unprivileged OpenVMS AXP PALcode Instruction Summary

Mnemonic	Operation and Description
REMQTIQR	Remove from quadword queue at tail, interlocked resident The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation. This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.
REMQUEL	Remove from longword queue The queue entry addressed by R16 for REMQUEL or the entry addressed by the longword addressed by R16 for REMQUEL/D is removed from the longword absolute queue, and the address of the removed entry is returned in R1. The removal is a noninterruptible operation.
REMQUEQ	Remove from quadword queue The queue entry addressed by R16 for REMQUEQ or the entry addressed by the quadword addressed by R16 for REMQUEL/D is removed from the quadword absolute queue, and the address of the removed entry removed is returned in R1. The removal is a noninterruptible operation.
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 AXP Palcode Instructions

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

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

Table 9–2: Privileged OpenVMS AXP 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 The HALT instruction 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.
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.

Table 9–2 (Cont.): Privileged OpenVMS AXP PALcode Instructions Summary

Mnemonic	Operation and Description
SWPCTX	Swap privileged context The SWPCTX instruction returns ownership of the data structure that contains the current hardware privileged context (the HWPCB) to the operating system and passes ownership of the new HWPCB to the processor.
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.

Chapter 10

DEC OSF/1

The following sections specifiy the Privileged Architecture Library (PALcode) instructions that are required to support a DEC OSF/1 system.

10.1 Unprivileged DEC OSF/1 PALcode Instructions

Table 10–1 describes the unprivileged DEC OSF/1 PALcode instructions.

Table 10–1: Unprivileged DEC OSF/1 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.
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.
rdunique	Read unique The rdunique instruction returns the process unique value.
wrunique	Write unique The wrunique instruction sets the process unique register.

10.2 Privileged DEC OSF/1 PALcode Instructions

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

Table 10–2 describes the privileged DEC OSF/1 PALcode instructions.

Table 10–2: Privileged DEC OSF/1 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.
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.

Table 10–2 (Cont.): Privileged DEC OSF/1 PALcode Instruction Summary

Mnemonic	Operation and Description
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.
wripr	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.

Table 10–2 (Cont.): Privileged DEC OSF/1 PALcode Instruction Summary

Mnemonic	Operation and Description
wrusp	Write user stack pointer The wrusp instruction writes a value to the user stack pointer while in kernel mode.
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).

Chapter 11
Windows NT AXP

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

11.1 Unprivileged Windows NT AXP PALcode Instructions

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 AXP 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.
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.

Table 11–1 (Cont.): Unprivileged Windows NT AXP PALcode Instruction Summary

Mnemonic	Operation and description
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 AXP PALcode Instruction Summary

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

Table 11–2: Privileged Windows NT AXP 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.
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.
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.
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.

Table 11–2 (Cont.): Privileged Windows NT AXP PALcode Instruction Summary

Mnemonic	Operation and description
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.
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.
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 (Cont.): Privileged Windows NT AXP PALcode Instruction Summary

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.
tbia	Translation buffer invalidate all The tbia instruction invalidates all translations and virtual cache blocks within the processor.

Table 11–2 (Cont.): Privileged Windows NT AXP PALcode Instruction Summary

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

Many of the performance optimizations discussed below are advantageous only for frequently executed code. For rarely executed code, they may produce a bigger program that is not any faster. Some of the branching optimizations also depend on good prediction of which path from a conditional branch is more frequently executed. These optimizations are best done by using an execution profile, either an estimate generated by compiler heuristics, or a real profile of a previous run, such as that gathered by PC-sampling in PCA.

Each computer architecture has a ‘‘natural word size.’’ For the PDP–11, it is 16 bits; for VAX, 32 bits; and for Alpha AXP, 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 AXP, 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 AXP 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 AXP architecture will have many implementations, an exact cache design cannot be outlined here. Nonetheless, some expected bounds can be stated.

1. Small (first-level) cache sizes will likely be in the range 2 KB to 64 KB
2. Small cache block sizes will likely be 16, 32, 64, or 128 bytes
3. Large (second- or third-level) cache sizes will likely be in the range 128 KB to 8 MB
4. Large cache block sizes will likely be 32, 64, 128, or 256 bytes
5. TB sizes will likely be in the range 16 to 1024 entries

Thus, if two data items need to go in different cache blocks, it is desirable to make them at least 128 bytes apart (modulo 2 KB). Doing so creates a high probability of allowing both items to be in a small cache simultaneously for all Alpha AXP implementations.

In each case below, the performance implication is given by an order-of-magnitude number: 1, 3, 10, 30, or 100. A factor of 10 means that the performance difference being discussed will likely range from 3 to 30 across all Alpha AXP implementations.

A.2 Instruction-Stream Considerations

The following sections describe considerations for the instruction stream.

A.2.1 Instruction Alignment

Code PSECTs should be octaword aligned. Targets of frequently taken branches should be at least quadword aligned, and octaword aligned for very frequent loops. Compilers could use execution profiles to identify frequently taken branches.

Most Alpha AXP implementations will fetch aligned quadwords of instruction stream (two instructions), and many will waste an instruction-issue cycle on a branch to an odd longword. High-end implementations may eventually fetch aligned octawords, and waste up to three issue cycles on a branch to an odd longword. Some implementations may only be able to fetch wide chunks of instructions every other CPU cycle. Fetching four instructions from an aligned octaword can get at most one cache miss, while fetching them from an odd longword address can get two or even three cache misses.

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

A.2.2 Multiple Instruction Issue — Factor of 3

Some Alpha AXP implementations will issue multiple instructions in a single cycle. To improve the odds of multiple-issue, compilers should choose pairs of instructions to put in aligned quadwords. Pick one from column A and one from column B (but only a total of one load/store/branch per pair).

Column A	Column B
Integer Operate	Floating Operate
Floating Load/Store	Integer Load/Store
Floating Branch	Integer Branch
	BR/BSR/JSR

Implementors of multiple-issue machines should give first priority to dual-issuing at least the above pairs, and second priority to multiple-issue of other combinations.

In general, the above rules will give a good hardware-software match, but compilers may want to implement model-specific switches to generate code tuned more exactly to a specific implementation.

A.2.3 Branch Prediction and Minimizing Branch-Taken — Factor of 3

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

Compilers should follow these rules to minimize unexpected branches:

1. Implementations will predict all forward conditional branches as not-taken, and all backward conditional branches as taken. Based on execution profiles, compilers should physically rearrange code so that it has matching behavior.
2. Make basic blocks as big as possible. A good goal is 20 instructions on average between branch-taken. This means unrolling loops so that they contain at least 20 instructions, and putting subroutines of less than 20 instructions directly in line. It also means using execution profiles to rearrange code so that the frequent case of a conditional branch falls through. For very high-performance loops, it will be profitable to move instructions across conditional branches to fill otherwise wasted instruction issue slots, even if the instructions moved will not always do useful work. Note that the Conditional Move instructions can sometimes be used to avoid breaking up basic blocks.
3. In an if-then-else construct whose execution profile is skewed even slightly away from 50%-50% (51-49 is enough), put the infrequent case completely out of line, so that the frequent case encounters *zero* branch-takens, and the infrequent case encounters *two* branch-takens. If the infrequent case is rare (5%), put it far enough away that it never comes into the I-cache. If the infrequent case is extremely rare (error message code), put it on a page of rarely executed code and expect that page *never* to be paged in.
4. There are two functionally identical branch-format opcodes, BSR and BR.

31–26	25–21	20–0	
BSR	Ra	Displacement	Branch Format
BR	Ra	Displacement	Branch Format

Compilers should use the first one for subroutine calls, and the second for GOTOs. Some implementations may push a stack of predicted return addresses for BSR and not push the stack for BR. Failure to compile the correct opcode will result in mispredicted return addresses, and hence make subroutine returns slow.

5. The memory-format JSR instruction has 16 unused bits. These should be used by the compilers to communicate a hint about expected branch-target behavior (see *Common Architecture, Chapter 4*).

31		16	15–0	
JSR	Ra	Rb		Memory Format

If the JSR is used for a computed GOTO or a CASE statement, compile bits <15:14> as 00, and bits <13:0> such that (updated PC+Instr<13:0>*4) <15:0> equals (likely_target_addr) <15:0>. In other words, pick the low 14 bits so that a normal PC+displacement*4 calculation will match the low 16 bits of the most likely target longword address. (Implementations will likely prefetch from the matching cache block.)

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

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

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

Implementors should give first priority to executing straight-line code with no branch-takens as quickly as possible, second priority to predicting conditional branches based on the sign of the displacement field (backward taken, forward not-taken), and third priority to predicting subroutine return addresses by running a small prediction stack. (VAX traces show a stack of two to four entries correctly predicts most branches.)

A.2.4 Improving I-Stream Density — Factor of 3

Compilers should try to use profiles to make sure almost 100% of the bytes brought into an I-cache are actually executed. This means aligning branch targets and putting rarely executed code out of line. Doing so would consistently make an I-cache appear about two times larger, compared to current VAX practice.

The example below shows the bytes actually brought into a VAX cache (from part of an address trace of a DLINPAC). The dots represent bytes brought into the cache but never executed. They occupy about half of the cache.

Each line shows the use of an aligned 64-byte I-cache block. A portion of DLINPAC and a portion of OpenVMS AXP 4.x are shown. Uppercase I is the first byte of an instruction, and lowercase i marks subsequent bytes. Period (.) shows a byte brought into the cache but never executed.

```
I-fetch  Byte 0                                                    Byte 63

-------- ----------------------------------------------------------------
000268C0 ........................IiiiIiiIiiIiiiiiiiiiIiii................
00026900 ................................................IiiiiIiiiiiiiiii
00026940 IiIiiIiIiiIiIiIiIiiiIiIiiIiIiiiiiiiIiiIiii......................
00026980 ........................................IiiiIiiIiiIiiIiiIiIiiIii
000269C0 I..............IiiiiIiiIiiiiIiIiiiiIiiiIIiIiIiiIiIiiiIiIiii.....
00026A00 ............................IiIiiiiiiiiiiiiiIiiIiiiIiii.........
00026A40 .................................IiiiiiiiiIiiiiiiiiIiIiiiIiiIii
00026A80 IiIiiiiIiIiIiiiIiIiIiIiiiiiiiiIiiIiiiIiii.................IiiIii
00026AC0 IiiIiii.........................................................

80004440 .............................................IiiiIiIiii.........
80004680 ....IiiiiiIiii..................................................
80004900 ................IiiIiiIiiIiiiiIiIiiIiiIiiIiiiIiIiiiiIiIiiiIiiiiI
80004940 IiiiiIiiiIiiIiIiii.............IiiiiIiii........................
80004A00 .................................................IiiiiiIiiIiiiii
80004A40 IiIiiIiiiiIiiiIiiiIiiiIiii............IiiiiiIIiiiiiIiiiiIiiIiiiI
80004A80 IiiiiIiiiIiiIiiIiiii....IiiIiiIiii...............................
80004F40 .............................IiiiiiIiiiiiiIiiiIiiiiiiIiii.......
80004F80 .......................IiiiIiiiiiiiIiiIiIiiiIiiiiiiiiiiiiiiIiiiI
80004FC0 IIiiiiiIiiiIiIiiiIiii.....IiiiiIiIiii...........................
80008A40 ........................................................IiiiIiii
80008A80 IIiiIiiiIiiIiIiiiIiIiIiiiIiIiiIiiiiiIiiIiiIiiIiiiiiiiIiIiiiIiii.
```

A.2.5 Instruction Scheduling — Factor of 3

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

''Result latency'' is defined as the number of CPU cycles that must elapse between an instruction that writes a result register and one that uses that register, if execution-time stalls are to be avoided. Thus, a latency of zero means that the instruction writes a result register and the instruction that uses that register can be multiple-issued in the *same* cycle. A latency of 2 means that if the writing instruction is issued at cycle N, the reading instruction can issue no earlier than cycle N+2. Latency is implementation specific.

Most Alpha AXP instructions have a non-zero result latency. Compilers should schedule code so that a result is not used too soon, at least in frequently executed code (inner loops, as identified by execution profiles). In general, this will require loop unrolling and short procedure inlining.

Assume that implementations can dual-issue instructions. Assume that Load and JSR instructions have a latency of 3, shifts and byte manipulation a latency of 2, integer multiply a latency of 10, and other integer operates a latency of 1. Assume floating multiply has a latency of 5, floating divide a latency of 10, and other floating operates a latency of 4. Scheduling to these latencies gives at least reasonable performance on current implementations.

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

Implementors should give first priority to minimizing the latency of back-to-back integer operations, of address calculations immediately followed by load/store, of load immediately followed by branch, and of compare immediately followed by branch. Second priority should be given to minimizing latencies in general.

A.3 Data-Stream Considerations

The following sections describe considerations for the data stream.

A.3.1 Data Alignment — Factor of 10

Data PSECTs should be at least octaword aligned, so that aggregates (arrays, some records, subroutine stack frames) can be allocated on aligned octaword boundaries to take advantage of any implementations with aligned octaword data paths, and to decrease the number of cache fills in almost all implementations.

Aggregates (arrays, records, common blocks, and so forth) should be allocated on at least aligned octaword boundaries whenever language rules allow. In some implementations, a series of writes that completely fill a cache block may be a factor of 10 faster than a series of writes that partially fill a cache block, when that cache block would give a read miss. This is true of write-back caches that read a partially filled cache block from memory, but optimize away the read for completely filled blocks.

For such implementations, long strings of sequential writes will be faster if they start on a cache-block boundary (a multiple of 128 bytes will do well for most, if not all, Alpha AXP implementations). This applies to array results that sweep through large portions of memory, and also to register-save areas for context switching, graphics frame buffer accesses, and other places where exactly 8, 16, 32, or more quadwords are stored sequentially. Allocating the targets at multiples of 8, 16, 32, or more quadwords, respectively, and doing the writes in order of increasing address will maximize the write speed.

Items within aggregates that are forced to be unaligned (records, common blocks) should generate compile-time warning messages and inline byte extract/insert code. Users must be educated that the warning message means that they are taking a factor of 30 performance hit.

Compilers should consider supplying a switch that allows the compiler to pad aggregates to avoid unaligned data.

Compiled code for parameters should assume that the parameters are aligned. Unaligned actuals will therefore cause run-time alignment traps and very slow fixups. The fixup routine, if invoked, should generate warning messages to the user, preferably giving the first few statement numbers that are doing unaligned parameter access, and at the end of a run the total number of alignment traps (and perhaps an estimate of the performance improvement if the data were aligned). Again, users must be educated that the trap routine warning message means they are taking a factor of 30 performance hit.

Frequently used scalars should reside in registers. Each scalar datum allocated in memory should normally be allocated an aligned quadword to itself, even if the datum is only a byte wide. This allows aligned quadword loads and stores and avoids partial-quadword writes (which may be half as fast as full-quadword writes, due to such factors as read-modify-write a quadword to do quadword ECC calculation).

Implementors should give first priority to fast reads of aligned octawords and second priority to fast writes of full cache blocks. Partial-quadword writes need not have a fast repetition rate.

A.3.2 Shared Data in Multiple Processors — Factor of 3

Software locks are aligned quadwords and should be allocated to large cache blocks that either contain no other data, or read-mostly data whose usage is correlated with the lock.

Whenever there is high contention for a lock, one processor will have the lock and be using the guarded data, while other processors will be in a read-only spin loop on the lock bit. Under these circumstances, *any* write to the cache block containing the lock will likely cause excess bus traffic and cache fills, thus having a performance impact on all processors that are involved, and the buses between them. In some decomposed FORTRAN programs, refills of the cache blocks containing one or two frequently used locks can account for a third of all the bus bandwidth the program consumes.

Whenever there is almost no contention for a lock, one processor will have the lock and be using the guarded data. Under these circumstances, it might be desirable to keep the guarded data in the *same* cache block as the lock.

For the high-sharing case, compilers should assume that *almost all* accesses to shared data result in cache misses all the way back to main memory, for each distinct cache block used. Such accesses will likely be a factor of 30 slower than cache hits. It is helpful to pack correlated shared data into a small number of cache blocks. It is helpful also to segregate blocks written by one processor from blocks read by others.

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

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

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

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

Similarly, operating-system scheduling algorithms should attempt to enforce some affinity between a given device's interrupts and the processor on which the interrupt-handler runs. I/O control data structures and locks for different devices should be disjoint. Doing both of these allows higher cache hit rates on the corresponding I/O control data structures.

Implementors should give first priority to an efficient (low-bandwidth) way of transferring isolated lock values and other isolated, shared write data between processors.

Implementors should assume that the amount of shared data will continue to increase, so over time the need for efficient sharing implementations will also increase.

A.3.3 Avoiding Cache/TB Conflicts — Factor of 1

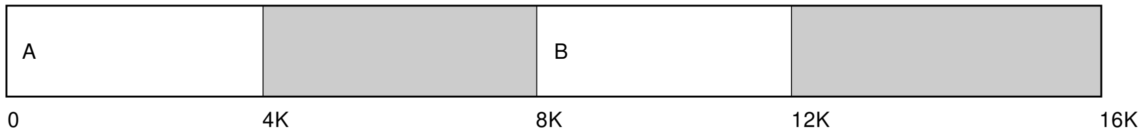
Occasionally, programs that run with a direct-mapped cache or TB will thrash, taking excessive cache or TB misses. With some work, thrashing can be minimized at compile time.

In a frequently executed loop, compilers could allocate the data items accessed from memory so that, on each loop iteration, all of the memory addresses accessed are either in *exactly the same* aligned 64-byte block, or differ in bits VA<10:6>. For loops that go through arrays in a common direction with a common stride, this means allocating the arrays, checking that the first-iteration addresses differ, and if not, inserting up to 64 bytes of padding *between* the arrays. This rule will avoid thrashing in small direct-mapped data caches with block sizes up to 64 bytes and total sizes of 2K bytes or more.

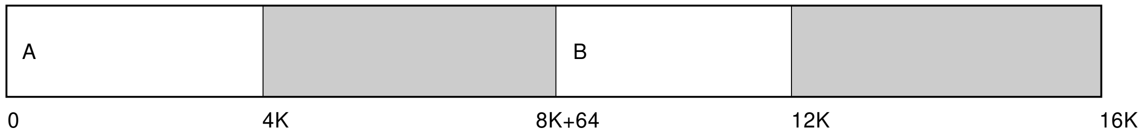
Example:

```
      REAL*4 A(1000),B(1000)
       DO 60 i=1,1000
   60 A(i) = f(B(i))
```

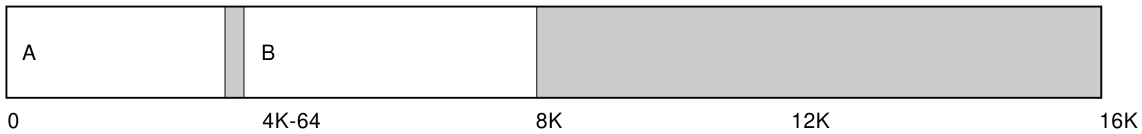
BAD allocation (A and B thrash in 8 KB direct-mapped cache):



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):



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):



In a frequently executed loop, compilers could allocate the data items accessed from memory so that, on each loop iteration, all of the memory addresses accessed are either in *exactly the same 8 KB* page, or differ in bits VA<17:13>. For loops that go through arrays in a common direction with a common stride, this means allocating the arrays, checking that the first-iteration addresses differ, and if not, inserting up to 8K bytes of padding *between* the arrays. This rule will avoid thrashing in direct-mapped TBs and in some large direct-mapped data caches, with total sizes of 32 pages (256 KB) or more.

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

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

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

A.3.4 Sequential Read/Write — Factor of 1

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

To use FETCH and FETCH_M effectively, software should follow this programming model:

1. Assume that at most two FETCH instructions can be outstanding at once, and that there are two prefetch address registers, PREa and PREb, to hold prefetching state. FETCH instructions alternate between loading PREa and PREb. Each FETCH instruction overwrites any previous prefetching state, thus terminating any previous prefetch that is still in progress in the register that is loaded. The order of fetching within a block and the order between PREa and PREb are UNPREDICTABLE.

   Implementation Note:

   Implementations are encouraged to alternate at convenient intervals between PREa and PREb.

2. Assume, for maximum efficiency, that there should be about 64 unrelated memory access instructions (load or store) between a FETCH and the first actual data access to the prefetched data.

3. Assume, for instruction-scheduling purposes in a multilevel cache hierarchy, that FETCH does not prefetch data to the innermost cache level, but rather one level out. Schedule loads to bury the last level of misses.

4. Assume that FETCH is worthwhile if, on average, at least half the data in a block will be accessed. Assume that FETCH_M is worthwhile if, on average, at least half the data in a block will be modified.

5. Treat FETCH as a vector load. If a piece of code could usefully prefetch four operands, launch the first two prefetches, do about 128 memory references worth of work, then launch the next two prefetches, do about 128 more memory references worth of work, then start using the four sets of prefetched data.

6. Treat FETCH as having the same effect on a cache as a series of 64 quadword loads. If the loads would displace useful data, so will FETCH. If two sets of loads

from specific addresses will thrash in a direct-mapped cache, so will two FETCH instructions using the same pair of addresses.

Implementation Note:

Hardware implementations are expected to provide either no support for FETCHx or support that closely matches this model.

A.4 Code Sequences

The following section describes code sequences.

A.4.1 Aligned Byte/Word Memory Accesses

The instruction sequences given in *Common Architecture, Chapter 4*, for byte and word accesses are worst-case code. 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, other divisions by constants can be closely approximated via multiplication by the reciprocal.

Integer division does not exist as a hardware opcode. Division by a constant can always be done via UMULH of another appropriate constant, followed by a right shift. General quadword division by true variables can be done via a subroutine. The subroutine could test for small divisors (less than about 1000 in absolute value) and for those, do a table lookup on the exact constant and shift count for an UMULH /shift sequence. For the remaining cases, a table lookup on about a 1000-entry table and a multiply can give a linear approximation to 1/divisor that is accurate to 16 bits.

Using this approximation, a multiply and a back-multiply and a subtract can generate one 16-bit quotient ‘‘digit’’ plus a 48-bit new partial dividend. Three more such steps can generate the full quotient. Having prior knowledge of the possible sizes of the divisor and dividend, normalizing away leading bytes of zeros, and performing an early-out test can reduce the average number of multiplies to about five (compared to a best case of one and a worst case of nine).

A.4.3 Byte Swap

When it is necessary to swap all the bytes of a datum, perhaps because the datum originated on a machine of the opposite byte numbering convention, the simplest sequence is to use the VAX floating-point load instruction to swap words, followed by an integer sequence to swap 4 pairs of bytes. Assume 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 "." in the comments stands for a byte of zeros (similar sequences can be used for data in registers, sometimes doing the byte swaps first and word swap second):

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

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

A.4.4 Stylized Code Forms

Using the same stylized code form for a common operation makes compiler output a little more readable and makes it more likely that an implementation will speed up the stylized form.

A.4.4.1 NOP

The universal NOP form is:

```
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 AXP literal construct in Operate instructions creates a canonical longword constant for values 0..255.

A longword constant stored in an Alpha AXP 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 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 Trap Barrier

The TRAPB instruction guarantees that it and any following instructions do not issue until all possible preceding traps have been signaled. This does not mean that all preceding instructions have necessarily run to completion (for example, a Load instruction may have passed all the fault checks but not yet delivered data from a cache miss).

A.4.6 Pseudo-Operations (Stylized Code Forms)

This section summarizes the pseudo-operations for the Alpha AXP architecture that may be used by various software components in an Alpha AXP system. Most of these forms are discussed in preceding sections.

In the context of this section, pseudo-operations all represent a single underlying machine instruction. Each pseudo-operation represents a particular instruction with either replicated fields (such as FMOV), or hard-coded zero fields. Since the pattern is distinct, these pseudo-operations can be decoded by instruction decode mechanisms.

In Table A–1, the pseudo-operation codes can be viewed as macros with parameters. The formal form is listed in the left column, and the expansion in the code stream listed in the right column.

Some instruction mnemonics have synonyms. These are different from pseudo-operations in that each synonym represents the same underlying instruction with no special encoding of operand fields. As a result, synonyms cannot be distinquished from each other. They are not listed in the table that follows. Examples of synonyms are: BIC/ANDNOT, BIS/OR, and EQV/XORNOT.

Table A–1: Decodable Pseudo-Operations (Stylized Code Forms)

Pseudo-Operation in Listing		Actual Instruction Encoding	
No-exception generic floating absolute value:			
FABS	Fx, Fy	CPYS	F31, Fx, Fy
Branch to target (21-bit signed displacement):			
BR	target	BR	R31, target
Clear integer register:			
CLR	Rx	BIS	R31, R31, Rx
Clear a floating-point register:			
FCLR	Fx	CPYS	F31, F31, Fx
Floating-point move:			
FMOV	Fx, Fy	CPYS	Fx, Fx, Fy
No-exception generic floating negation:			
FNEG	Fx, Fy	CPYSN	Fx, Fx, Fy
Floating-point no-op:			
FNOP		CPYS	F31, F31, F31
Move Rx/8-bit zero-extended literal to Ry:			
MOV	{Rx/Lit8}, Ry	BIS	R31, {Rx/Lit8}, Ry
Move 16-bit sign-extended literal to Rx:			
MOV	Lit, Rx	LDA	Rx, lit(R31)
Move to FPCR:			
MT_FPCR	Fx	MT_FPCR	Fx, Fx, Fx
Move from FPCR:			
MF_FPCR	Fx	MF_FPCR	Fx, Fx, Fx
Negate F_floating:			
NEGF	Fx, Fy	SUBF	F31, Fx, Fy
Negate F_floating, semi-precise:			
NEGF/S	Fx, Fy	SUBF/S	F31, Fx, Fy
Negate G_floating:			
NEGG	Fx, Fy	SUBG	F31, Fx, Fy
Negate G_floating, semi-precise:			
NEGG/S	Fx, Fy	SUBG/S	F31, Fx, Fy
Negate longword:			

Table A–1 (Cont.): Decodable Pseudo-Operations (Stylized Code Forms)

Pseudo-Operation in Listing		Actual Instruction Encoding	
NEGL	{Rx/Lit8}, Ry	SUBL	R31, {Rx/Lit}, Ry
Negate longword with overflow detection:			
NEGL/V	{Rx/Lit8}, Ry	SUBL/V	R31, {Rx/Lit}, Ry
Negate quadword:			
NEGQ	{Rx/Lit8}, Ry	SUBQ	R31, {Rx/Lit}, Ry
Negate quadword with overflow detection:			
NEGQ/V	{Rx/Lit8}, Ry	SUBQ/V	R31, {Rx/Lit}, Ry
Negate S_floating:			
NEGS	Fx, Fy	SUBS	F31, Fx, Fy
Negate S_floating, software with underflow detection:			
NEGS/SU	Fx, Fy	SUBS/SU	F31, Fx, Fy
Negate S_floating, software with underflow and inexact result detection:			
NEGS/SUI	Fx, Fy	SUBS/SUI	F31, Fx, Fy
Negate T_floating:			
NEGT	Fx, Fy	SUBT	F31, Fx, Fy
Negate T_floating, software with underflow detection:			
NEGT/SU	Fx, Fy	SUBT/SU	F31, Fx, Fy
Negate T_floating, software with underflow and inexact result detection:			
NEGT/SUI		SUBT/SUI	F31, Fx, Fy
Integer no-op:			
NOP		BIS	R31, R31, R31
Logical NOT of Rx/8-bit zero-extended literal storing results in Ry:			
NOT	{Rx/Lit8}, Ry	ORNOT	R31, {Rx/Lit}, Ry
Longword sign-extension of Rx storing results in Ry:			
SEXTL	{Rx/Lit8}, Ry	ADDL	R31, {Rx/Lit}, Ry
Universal NOP for both integer and floating-point code:			
UNOP		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 AXP implementations:

1. At most 40 operate or conditional-branch (not taken) instructions executed in the sequence between LDx_L and STx_C.
2. At most two I-stream TB-miss faults. Sequential instruction execution guarantees this.
3. No other exceptions triggered during the last execution of the sequence.

Implementation Note:

On all expected implementations, this allows for about 50 μsec of execution time, even with 100 percent cache misses. This should satisfy any requirement for a 1 msec timer interrupt rate.

Appendix B

IEEE Floating-Point Conformance

A subset of IEEE Standard for Binary Floating-Point Arithmetic (754-1985) is provided in the Alpha AXP 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 AXP Choices for IEEE Options

Alpha AXP supports IEEE single, double, and optionally (in software) extended double formats. There is no hardware support for the optional extended double format.

Alpha AXP hardware supports normal and chopped IEEE rounding modes. IEEE plus infinity and minus infinity rounding modes can be implemented in hardware or software.

Alpha AXP hardware does not support optional IEEE software trap enable/disable modes; see the following discussion about software support.

Alpha AXP hardware supports add, subtract, multiply, divide, convert between floating formats, convert between floating and integer formats, and compare. Software routines support square root, remainder, round to integer in floating-point format, and convert binary to/from decimal.

In the Alpha AXP 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 AXP choice is that the accuracy provided by conversions between decimal strings and binary floating-point numbers will meet or exceed IEEE standard requirements. It is implementation-dependent whether the software binary/decimal conversions beyond 9 or 17 digits treat any excess digits as zeros.

Overflow and underflow, NaNs, and infinities encountered during software binary to decimal conversion return strings that specify the conditions. Such strings can be truncated to their shortest unambiguous length.

Alpha AXP hardware supports comparisons of same-format numbers. Software supports comparisons of different-format numbers.

In the Alpha AXP architecture, results are true-false in response to a predicate.

Alpha AXP hardware supports the required six predicates and the optional unordered predicate. The other 19 optional predicates can be constructed from sequences of two comparisons and two branches.

Except for the compare instructions (CMPTyy) and the Overflow Disable (OVFD) option, Alpha AXP hardware supports infinity arithmetic by trapping. That is the case when an infinity operand is encountered and when an infinity is to be created from finite operands by overflow or division by zero. A software trap handler (interposed between the hardware and the IEEE user) provides correct infinity arithmetic.

Except for the Invalid Operation Disable (INVD) option, Alpha AXP hardware supports NaNs by trapping when a NaN operand is encountered and when a NaN is to be created. A software trap handler (interposed between the hardware and the IEEE user) provides correct Signaling and Quiet NaN behavior.

In the Alpha AXP architecture, Quiet NaNs do not afford retrospective diagnostic information.

In the Alpha AXP 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 AXP hardware fully supports negative zero operands, and follows the IEEE rules for creating negative zero results.

Except for the optional trap disable bits in the FPCR, Alpha AXP hardware does not supply IEEE exception trap behavior; the hardware traps are a superset of the IEEE-required conditions. A software trap handler (interposed between the hardware and the IEEE user) provides correct IEEE exception behavior.

In the Alpha AXP architecture, tininess is detected by hardware after rounding, and loss of accuracy is detected by software as an inexact result.

In the Alpha AXP architecture, user trap handlers are supported by compilers and a software trap handler (interposed between the hardware and the IEEE user), as described in the next section.

B.2 Alpha AXP Hardware Support of Software Exception Handlers

Except for the optional trap disable bits in the FPCR, the hardware trap behavior of Alpha AXP instructions is determined at compile time; short of recompiling, there are no dynamic facilities for changing hardware trap behavior.

B.2.1 Choosing Degrees of IEEE Compliance

There is an essential disparity between the Alpha AXP design goal of fast execution and the IEEE design goal of exact trap behavior. The Alpha AXP hardware architecture provides means for users to choose various degrees of IEEE compliance, at appropriate performance cost.

Instructions compiled without the /Software modifier cannot produce IEEE-compliant trap or status bit behavior, nor can they provide IEEE-compliant non-finite arithmetic. Trapping and stopping on non-finite operands or results (rather than the IEEE default of continuing with NaNs propagated) is an Alpha AXP value-added behavior that some users prefer.

Instructions compiled without the /Underflow hardware trap enable modifier cannot produce IEEE-compliant underflow trap or status bit behavior, nor can they provide IEEE-compliant denormal results. They are fast and provide true zero (not minus zero) results whenever underflow occurs. This is an Alpha AXP value-added behavior that some users prefer.

Instructions compiled without the /Inexact hardware trap enable modifier cannot produce IEEE-compliant inexact trap or status bit behavior. Except when the Inexact Disable (INED) option is implemented, trapping on inexact is painfully slow. Few users appear to prefer enabling inexact trapping, but they can get it if they really want it.

Except when the optional Overflow Disable (OVFD), Division by Zero Disable (DZED), or Invalid Operation Disable (INVD) bits in the FPCR are set, IEEE floating-point instructions compiled with the /Software enable modifier produce hardware traps and unpredictable values for overflow, division by zero, or invalid operation. A software trap handler may then produce the chosen IEEE-required behavior. The software trap handler reports an enabled IEEE exception to the user application as a fault, rather than as a trap. Because the exception is reported as a fault, the reported PC points to the trigger instruction, rather than to a point after the trigger instruction.

Regardless of whether or not an enabled fault occurs, the software completion handler sets the result register and the status flags to the IEEE standard nontrapping result, as further defined in the *IEEE Standard* section in *Common Architecture, Chapter 4*.

Except when the optional Underflow Disable (UNFD) bit in the FPCR is set, IEEE floating-point instructions compiled with the /Software enable and /Underflow enable modifiers, produce hardware traps and true zero values for underflow; a software trap handler may then produce all IEEE-required behavior. Such instructions with /Software and /Underflow enabled, but without an underflow condition that produce zero value results, always have the correct sign.

IEEE floating-point instructions compiled with the /Inexact enable modifier produce hardware traps that allow a software trap handler to produce all IEEE-required behavior.

Thus, to get full IEEE compliance of all the required features of the standard, users must compile with all three options enabled.

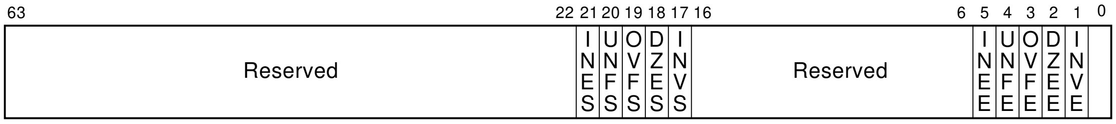
To get the optional full IEEE user trap handler behavior, a software trap handler must be provided that implements the five exception flags, dynamic user trap handler disabling, handler saving and restoring, default behavior for disabled user trap handlers, and linkages that allow a user handler to return a substitute result.

The software trap handler uses the FP_Control quadword, along with the floating-point control register (FPCR), to provide various levels of IEEE-compliant behavior.

B.2.2 IEEE Floating-Point Control (FP_C) Quadword

Operating system implementations provide the following support for an IEEE floating-point control quadword (FP_C), illustrated in Figure B–1 and described in Table B–1.

Figure B–1: IEEE Floating-Point Control (FP_C) Quadword



- The operating system software completion mechanism maintains the FP_C. Therefore, the FP_C affects (and is affected by) only those instructions with the /Software enable modifier.
- The FP_C quadword is context switched when the operating system switches the thread context. (Note that 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 integer overflow software completion mechanism.
- At process creation, all trap enable flags in the FP_C are clear. The setting 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.
47–22	Reserved for future architecture definition.
21	Inexact Result Status (INES) A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.

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

Bit	Description
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.
11-6	Reserved for future architecture definition.
5	Inexact Result Enable (INEE) Initiate an INE exception if a floating arithmetic or conversion operation gives a result that 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.

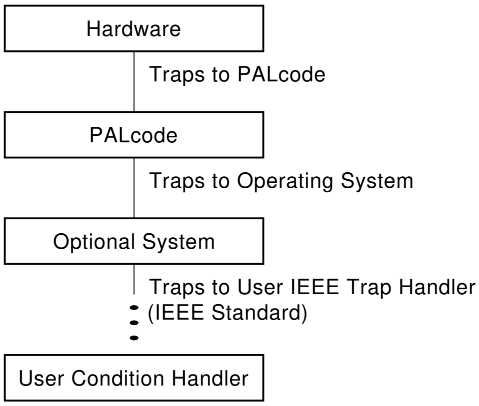
B.3 Mapping to IEEE Standard

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.

Our assumption, therefore, is that the only access to IEEE-specified software trap-enabled results will be generated in assembly language code. The following design allows this, but *only* if such assembly language code has TRAPB instructions after each floating-point instruction, and generates the IEEE-specified scaled result in a trap handler by emulating the instruction that was trapped by hardware overflow /underflow detection, using the original operands.

There is a set of detailed IEEE-specified result values, both for operations that are specified to raise IEEE traps and those that do not. This behavior is created on Alpha AXP by four layers of hardware, PALcode, the operating-system trap handler, and the user IEEE trap handler, as shown in Figure B–2.

Figure B–2: IEEE Trap Handling Behavior



The IEEE-specified trap behavior occurs *only* with respect to the user IEEE trap handler (the last layer in Figure B–2); any trap-and-fixup behavior in the first three layers is outside the scope of the IEEE standard.

The IEEE number system is divided into finite and non-finite numbers:

- The finites are normal numbers:

  –MAX..–MIN, –0, 0, +MIN..+MAX

- The non-finites are:

  Denormals, +/– Infinity, Signaling NaN, Quiet NaN

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

Table B–2 specifies, for the IEEE /Software modes, which layer does each piece of trap handling. See *Common Architecture, Chapter 4* for more detail on the hardware instruction descriptions.

Table B–2: IEEE Floating-Point Trap Handling

Alpha AXP Instructions	Hardware	PAL	OS Trap Handler	User Software 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	–
+/-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] Scale by bias adjust
Exponent underflow and disabled	Supply +0	–	–	–[1]
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by bias adjust
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply sum and trap	Trap	–	[Inexact]

[1] An implementation could choose instead to trap to PALcode and have the PALcode supply a zero result on all underflows.

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

Alpha AXP Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
MULx INPUT Exceptions				
Denormal operand	Trap	Trap	Supply prod.	–
+/-Inf operand	Trap	Trap	Supply prod.	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
0 * Inf	Trap	Trap	Supply QNaN	[Invalid Op]
MULx OUTPUT Exceptions				
Exponent overflow	Trap	Trap	Supply +/–Inf +/–MAX	[Overflow] Scale by bias adjust
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and Trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by bias adjust
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply prod. and trap	Trap	–	[Inexact]
DIVx INPUT Exceptions				
Denormal operand	Trap	Trap	Supply quot.	–
+/-Inf operand	Trap	Trap	Supply quot.	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]

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

Alpha AXP Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
DIVx INPUT Exceptions				
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] Scale by bias adjust
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by bias adjust
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply quot. and trap	Trap	–	[Inexact]
CMPTEQ CMPTUN INPUT Exceptions				
Denormal operand	Trap	Trap	Supply (=)	–
QNaN operand	Trap	Trap	Supply False for EQ, True for UN	–
SNaN operand	Trap	Trap	Supply False/ True	[Invalid Op]
CMPTLT CMPTLE INPUT Exceptions				
Denormal operand	Trap	Trap	Supply $\leq$ or $<$	—
QNaN operand	Trap	Trap	Supply False	[Invalid Op]

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

Alpha AXP Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
CMPTLT CMPTLE INPUT Exceptions				
SNaN operand	Trap	Trap	Supply False	[Invalid Op]
CVTfi INPUT Exceptions				
Denormal operand	Trap	Trap	Supply Cvt	–
+/-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][2]
CVTif OUTPUT Exceptions				
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply Cvt and trap	Trap	–	[Inexact]
CVTff INPUT Exceptions				
Denormal operand	Trap	Trap	Supply Cvt	–
+/-Inf operand	Trap	Trap	Supply Cvt	–

[2] An implementation could choose instead to trap to PALcode on extreme values and have the PALcode supply a truncated result on all overflows.

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

Alpha AXP Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
CVTff INPUT Exceptions				
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] Scale by bias adjust
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by bias adjust
Inexact and disabled	–	–	–	–
Inexact and enabled	Supply Cvt and trap	Trap	–	[Inexact]

Other IEEE operations (software subroutines or sequences of instructions), are listed here for completeness:

Remainder
SQRT
Round float to integer-valued float
Convert binary to/from decimal
Compare, other combinations than the four above

Table B–3 shows the IEEE standard charts.

Table B–3: IEEE Standard Charts

Exception	IEEE Software TRAP Disabled (IEEE Default)	IEEE Software TRAP 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(Inv, 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 contains a summary of all instructions and opcodes in the Alpha AXP architecture. All values are in hexadecimal radix.

C.1 Common Architecture Instruction Summary

This section contains a summary of all common Alpha AXP 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

Qualifiers for operate format instructions are shown in Table C–2. Qualifiers for IEEE and VAX floating-point instructions are shown in Sections C–3 and C–4, respectively.

Table C–2: Common Architecture Instructions

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
AND	Opr	11.00	Logical product
BEQ	Bra	39	Branch if $=$ zero
BGE	Bra	3E	Branch if $\geq$ zero
BGT	Bra	3F	Branch if $>$ zero
BIC	Opr	11.0	Bit clear
BIS	Opr	11.20	Logical sum
BLBC	Bra	38	Branch if low bit clear
BLBS	Bra	3C	Branch if low bit set
BLE	Bra	3B	Branch if $\leq$ zero
BLT	Bra	3A	Branch if $<$ zero
BNE	Bra	3D	Branch if $\neq$ zero
BR	Bra	30	Unconditional branch
BSR	Mbr	34	Branch to subroutine
CALL_PAL	Pcd	00	Trap to PALcode
CMOVEQ	Opr	11.24	CMOVE if $=$ zero
CMOVGE	Opr	11.46	CMOVE if $\geq$ zero
CMOVGT	Opr	11.66	CMOVE if $>$ zero
CMOVLBC	Opr	11.16	CMOVE if low bit clear
CMOVLBS	Opr	11.14	CMOVE if low bit set
CMOVLE	Opr	11.64	CMOVE if $\leq$ zero
CMOVLT	Opr	11.44	CMOVE if $<$ zero
CMOVNE	Opr	11.26	CMOVE if $\neq$ zero
CMPBGE	Opr	10.0F	Compare byte
CMPEQ	Opr	10.2D	Compare signed quadword equal
CMPGEQ	F-P	15.0A5	Compare G_floating equal
CMPGLE	F-P	15.0A7	Compare G_floating less than or equal
CMPGLT	F-P	15.0A6	Compare G_floating less than
CMPLE	Opr	10.6D	Compare signed quadword less than or equal
CMPLT	Opr	10.4D	Compare signed quadword less than
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

Table C–2 (Cont.): Common Architecture Instructions

Mnemonic	Format	Opcode	Description
CVTDG	F-P	15.09E	Convert D_floating to G_floating
CVTGD	F-P	15.0AD	Convert G_floating to D_floating
CVTGF	F-P	15.0AC	Convert G_floating to F_floating
CVTGQ	F-P	15.0AF	Convert G_floating to quadword
CVTLQ	F-P	17.010	Convert longword to quadword
CVTQF	F-P	15.0BC	Convert quadword to F_floating
CVTQG	F-P	15.0BE	Convert quadword to G_floating
CVTQL	F-P	17.030	Convert quadword to longword
CVTQL/SV		17.530	
CVTQL/V		17.130	
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
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 $\geq$ zero
FBGT	Bra	37	Floating branch if > zero
FBLE	Bra	33	Floating branch if $\leq$ zero
FBLT	Bra	32	Floating branch if < zero
FBNE	Bra	35	Floating branch if $\neq$ zero
FCMOVEQ	F-P	17.02A	FCMOVE if = zero
FCMOVGE	F-P	17.02D	FCMOVE if $\geq$ zero
FCMOVGT	F-P	17.02F	FCMOVE if > zero
FCMOVLE	F-P	17.02E	FCMOVE if $\leq$ zero
FCMOVLT	F-P	17.02C	FCMOVE if < zero
FCMOVNE	F-P	17.02B	FCMOVE if $\neq$ zero
FETCH	Mfc	18.8000	Prefetch data
FETCH_M	Mfc	18.A000	Prefetch data, modify intent
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

Table C–2 (Cont.): Common Architecture Instructions

Mnemonic	Format	Opcode	Description
INSWH	Opr	12.57	Insert word high
INSWL	Opr	12.1B	Insert word low
JMP	Mbr	1A.0	Jump
JSR	Mbr	1A.1	Jump to subroutine
JSR_COROUTINE	Mbr	1A.3	Jump to subroutine return
LDA	Mem	08	Load address
LDAH	Mem	09	Load address high
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
MB	Mfc	18.4000	Memory barrier
MF_FPCR	F-P	17.025	Move from FPCR
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
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
S8SUBQ	Opr	10.3B	Scaled subtract quadword by 8

Table C–2 (Cont.): Common Architecture Instructions

Mnemonic	Format	Opcode	Description
SLL	Opr	12.39	Shift left logical
SRA	Opr	12.3C	Shift right arithmetic
SRL	Opr	12.34	Shift right logical
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
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
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 these instructions is 16_{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							
CMPTLT	0A6							
CMPTLE	0A7							
CMPTUN	0A4							
CVTQS	0BC	03C	07C	0FC				
CVTQT	0BE	03E	07E	0FE				
CVTST	See below							
CVTTQ	See below							
CVTTS	0AC	02C	06C	0EC	1AC	12C	16C	1EC
DIVS	083	003	043	0C3	183	103	143	1C3
DIVT	0A3	023	063	0E3	1A3	123	163	1E3
MULS	082	002	042	0C2	182	102	142	1C2
MULT	0A2	022	062	0E2	1A2	122	162	1E2
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							
CMPTLT	5A6							
CMPTLE	5A7							
CMPTUN	5A4							
CVTQS					7BC	73C	77C	7FC
CVTQT					7BE	73E	77E	7FE
CVTTS	5AC	52C	56C	5EC	7AC	72C	76C	7EC
DIVS	583	503	543	5C3	783	703	743	7C3
DIVT	5A3	523	563	5E3	7A3	723	763	7E3
MULS	582	502	542	5C2	782	702	742	7C2
MULT	5A2	522	562	5E2	7A2	722	762	7E2
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 (Cont.): IEEE Floating-Point Instruction Function Codes

	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:

Since underflow cannot occur for CMPTxx, there is no difference in function or performance between CMPTxx/S and CMPTxx/SU. It is intended that software generate CMPTxx/SU in place of CMPTxx/S.

In the same manner, CVTQS and CTVQT can take an inexact result trap, but not an underflow. Because there is no encoding for a CVTQx/SI instruction, it is intended that software generate CVTQx/SUI in place of CVTQx/SI.

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 these instructions is 15_{16}.

Table C–4: VAX Floating-Point Instruction Function Codes

	None	/C	/U	/UC	/S	/SC	/SU	/SUC
ADDF	080	000	180	100	480	400	580	500
CVTDG	09E	01E	19E	11E	49E	41E	59E	51E
ADDG	0A0	020	1A0	120	4A0	420	5A0	520
CMPGEQ	0A5				4A5			
CMPGLT	0A6				4A6			
CMPGLE	0A7				4A7			
CVTGF	0AC	02C	1AC	12C	4AC	42C	5AC	52C
CVTGD	0AD	02D	1AD	12D	4AD	42D	5AD	52D
CVTGQ	See below							
CVTQF	0BC	03C						
CVTQG	0BE	03E						
DIVF	083	003	183	103	483	403	583	503
DIVG	0A3	023	1A3	123	4A3	423	5A3	523
MULF	082	002	182	102	482	402	582	502
MULG	0A2	022	1A2	122	4A2	422	5A2	522
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 Opcode Summary

Table C–5 lists all Alpha AXP 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 the all 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–5 are explained in Table C–6.

Table C–5: 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	Res	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	Res	Res	Res	STF (mem)	STL (mem)	BSR (br)	BLBS (br)
5/D	Res	Res	FLTV* (op)	\PAL\	STG (mem)	STQ (mem)	FBNE (br)	BNE (br)
6/E	Res	Res	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–6: Key to Opcode Summary (Table C–5)

Symbol	Meaning
FLTI*	IEEE floating-point instruction opcodes
FLTL*	Floating-point Operate instruction opcodes
FLTV*	VAX floating-point instruction opcodes
INTA*	Integer arithmetic instruction opcodes
INTL*	Integer logical instruction opcodes
INTM*	Integer multiply instruction opcodes
INTS*	Integer shift instruction opcodes
JSR*	Jump instruction opcodes
MISC*	Miscellaneous instruction opcodes
PAL*	PALcode instruction (CALL_PAL) opcodes
\PAL\	Reserved for PALcode
Res	Reserved for Digital

C.5 Common Architecture Opcodes in Numerical Order

Table C–7: Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
00	CALL_PAL	11.26	CMOVNE	15.01E	CVTDG/C
01	OPC01	11.28	ORNOT	15.020	ADDG/C
02	OPC02	11.40	XOR	15.021	SUBG/C
03	OPC03	11.44	CMOVLT	15.022	MULG/C
04	OPC04	11.46	CMOVGE	15.023	DIVG/C
05	OPC05	11.48	EQV	15.02C	CVTGF/C
06	OPC06	11.64	CMOVLE	15.02D	CVTGD/C
07	OPC07	11.66	CMOVGT	15.02F	CVTGQ/C
08	LDA	12.02	MSKBL	15.03C	CVTQF/C
09	LDAH	12.06	EXTBL	15.03E	CVTQG/C
0A	OPC0A	12.0B	INSBL	15.080	ADDF
0B	LDQ_U	12.12	MSKWL	15.081	SUBF
0C	OPC0C	12.16	EXTWL	15.082	MULF
0D	OPC0D	12.1B	INSWL	15.083	DIVF
0E	OPC0E	12.22	MSKLL	15.09E	CVTDG
0F	STQ_U	12.26	EXTLL	15.0A0	ADDG
10.00	ADDL	12.2B	INSLL	15.0A1	SUBG
10.02	S4ADDL	12.30	ZAP	15.0A2	MULG
10.09	SUBL	12.31	ZAPNOT	15.0A3	DIVG
10.0B	S4SUBL	12.32	MSKQL	15.0A5	CMPGEQ
10.0F	CMPBGE	12.34	SRL	15.0A6	CMPGLT
10.12	S8ADDL	12.36	EXTQL	15.0A7	CMPGLE
10.1B	S8SUBL	12.39	SLL	15.0AC	CVTGF
10.1D	CMPULT	12.3B	INSQL	15.0AD	CVTGD
10.20	ADDQ	12.3C	SRA	15.0AF	CVTGQ
10.22	S4ADDQ	12.52	MSKWH	15.0BC	CVTQF
10.29	SUBQ	12.57	INSWH	15.0BE	CVTQG
10.2B	S4SUBQ	12.5A	EXTWH	15.100	ADDF/UC
10.2D	CMPEQ	12.62	MSKLH	15.101	SUBF/UC
10.32	S8ADDQ	12.67	INSLH	15.102	MULF/UC
10.3B	S8SUBQ	12.6A	EXTLH	15.103	DIVF/UC
10.3D	CMPULE	12.72	MSKQH	15.11E	CVTDG/UC
10.40	ADDL/V	12.77	INSQH	15.120	ADDG/UC
10.49	SUBL/V	12.7A	EXTQH	15.121	SUBG/UC
10.4D	CMPLT	13.00	MULL	15.122	MULG/UC
10.60	ADDQ/V	13.20	MULQ	15.123	DIVG/UC
10.69	SUBQ/V	13.30	UMULH	15.12C	CVTGF/UC
10.6D	CMPLE	13.40	MULL/V	15.12D	CVTGD/UC
11.00	AND	13.60	MULQ/V	15.12F	CVTGQ/VC
11.08	BIC	14	OPC14	15.180	ADDF/U
11.14	CMOVLBS	15.000	ADDF/C	15.181	SUBF/U
11.16	CMOVLBC	15.001	SUBF/C	15.182	MULF/U
11.20	BIS	15.002	MULF/C	15.183	DIVF/U
11.24	CMOVEQ	15.003	DIVF/C	15.19E	CVTDG/U

Table C–7 (Cont.): Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
15.1A0	ADDG/U	15.580	ADDF/SU	16.0A6	CMPTLT
15.1A1	SUBG/U	15.581	SUBF/SU	16.0A7	CMPTLE
15.1A2	MULG/U	15.582	MULF/SU	16.0AC	CVTTS
15.1A3	DIVG/U	15.583	DIVF/SU	16.0AF	CVTTQ
15.1AC	CVTGF/U	15.59E	CVTDG/SU	16.0BC	CVTQS
15.1AD	CVTGD/U	15.5A0	ADDG/SU	16.0BE	CVTQT
15.1AF	CVTGQ/V	15.5A1	SUBG/SU	16.0C0	ADDS/D
15.400	ADDF/SC	15.5A2	MULG/SU	16.0C1	SUBS/D
15.401	SUBF/SC	15.5A3	DIVG/SU	16.0C2	MULS/D
15.402	MULF/SC	15.5AC	CVTGF/SU	16.0C3	DIVS/D
15.403	DIVF/SC	15.5AD	CVTGD/SU	16.0E0	ADDT/D
15.41E	CVTDG/SC	15.5AF	CVTGQ/SV	16.0E1	SUBT/D
15.420	ADDG/SC	16.000	ADDS/C	16.0E2	MULT/D
15.421	SUBG/SC	16.001	SUBS/C	16.0E3	DIVT/D
15.422	MULG/SC	16.002	MULS/C	16.0EC	CVTTS/D
15.423	DIVG/SC	16.003	DIVS/C	16.0EF	CVTTQ/D
15.42C	CVTGF/SC	16.020	ADDT/C	16.0FC	CVTQS/D
15.42D	CVTGD/SC	16.021	SUBT/C	16.0FE	CVTQT/D
15.42F	CVTGQ/SC	16.022	MULT/C	16.100	ADDS/UC
15.480	ADDF/S	16.023	DIVT/C	16.101	SUBS/UC
15.481	SUBF/S	16.02C	CVTTS/C	16.102	MULS/UC
15.482	MULF/S	16.02F	CVTTQ/C	16.103	DIVS/UC
15.483	DIVF/S	16.03C	CVTQS/C	16.120	ADDT/UC
15.49E	CVTDG/S	16.03E	CVTQT/C	16.121	SUBT/UC
15.4A0	ADDG/S	16.040	ADDS/M	16.122	MULT/UC
15.4A1	SUBG/S	16.041	SUBS/M	16.123	DIVT/UC
15.4A2	MULG/S	16.042	MULS/M	16.12C	CVTTS/UC
15.4A3	DIVG/S	16.043	DIVS/M	16.12F	CVTTQ/VC
15.4A5	CMPGEQ/S	16.060	ADDT/M	16.140	ADDS/UM
15.4A6	CMPGLT/S	16.061	SUBT/M	16.141	SUBS/UM
15.4A7	CMPGLE/S	16.062	MULT/M	16.142	MULS/UM
15.4AC	CVTGF/S	16.063	DIVT/M	16.143	DIVS/UM
15.4AD	CVTGD/S	16.06C	CVTTS/M	16.160	ADDT/UM
15.4AF	CVTGQ/S	16.06F	CVTTQ/M	16.161	SUBT/UM
15.500	ADDF/SUC	16.07C	CVTQS/M	16.162	MULT/UM
15.501	SUBF/SUC	16.07E	CVTQT/M	16.163	DIVT/UM
15.502	MULF/SUC	16.080	ADDS	16.16C	CVTTS/UM
15.503	DIVF/SUC	16.081	SUBS	16.16F	CVTTQ/VM
15.51E	CVTDG/SUC	16.082	MULS	16.180	ADDS/U
15.520	ADDG/SUC	16.083	DIVS	16.181	SUBS/U
15.521	SUBG/SUC	16.0A0	ADDT	16.182	MULS/U
15.522	MULG/SUC	16.0A1	SUBT	16.183	DIVS/U
15.523	DIVG/SUC	16.0A2	MULT	16.1A0	ADDT/U
15.52C	CVTGF/SUC	16.0A3	DIVT	16.1A1	SUBT/U
15.52D	CVTGD/SUC	16.0A4	CMPTUN	16.1A2	MULT/U
15.52F	CVTGQ/SVC	16.0A5	CMPTEQ	16.1A3	DIVT/U

Table C–7 (Cont.): Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
16.1AC	CVTTS/U	16.5AF	CVTTQ/SV	16.7BC	CVTQS/SUI
16.1AF	CVTTQ/V	16.5C0	ADDS/SUD	16.7BE	CVTQT/SUI
16.1C0	ADDS/UD	16.5C1	SUBS/SUD	16.7C0	ADDS/SUID
16.1C1	SUBS/UD	16.5C2	MULS/SUD	16.7C1	SUBS/SUID
16.1C2	MULS/UD	16.5C3	DIVS/SUD	16.7C2	MULS/SUID
16.1C3	DIVS/UD	16.5E0	ADDT/SUD	16.7C3	DIVS/SUID
16.1E0	ADDT/UD	16.5E1	SUBT/SUD	16.7E0	ADDT/SUID
16.1E1	SUBT/UD	16.5E2	MULT/SUD	16.7E1	SUBT/SUID
16.1E2	MULT/UD	16.5E3	DIVT/SUD	16.7E2	MULT/SUID
16.1E3	DIVT/UD	16.5EC	CVTTS/SUD	16.7E3	DIVT/SUID
16.1EC	CVTTS/UD	16.5EF	CVTTQ/SVD	16.7EC	CVTTS/SUID
16.1EF	CVTTQ/VD	16.6AC	CVTST/S	16.7EF	CVTTQ/SVID
16.2AC	CVTST	16.700	ADDS/SUIC	16.7FC	CVTQS/SUID
16.500	ADDS/SUC	16.701	SUBS/SUIC	16.7FE	CVTQT/SUID
16.501	SUBS/SUC	16.702	MULS/SUIC	17.010	CVTLQ
16.502	MULS/SUC	16.703	DIVS/SUIC	17.020	CPYS
16.503	DIVS/SUC	16.720	ADDT/SUIC	17.021	CPYSN
16.520	ADDT/SUC	16.721	SUBT/SUIC	17.022	CPYSE
16.521	SUBT/SUC	16.722	MULT/SUIC	17.024	MT_FPCR
16.522	MULT/SUC	16.723	DIVT/SUIC	17.025	MF_FPCR
16.523	DIVT/SUC	16.72C	CVTTS/SUIC	17.02A	FCMOVEQ
16.52C	CVTTS/SUC	16.72F	CVTTQ/SVIC	17.02B	FCMOVNE
16.52F	CVTTQ/SVC	16.73C	CVTQS/SUIC	17.02C	FCMOVLT
16.540	ADDS/SUM	16.73E	CVTQT/SUIC	17.02D	FCMOVGE
16.541	SUBS/SUM	16.740	ADDS/SUIM	17.02E	FCMOVLE
16.542	MULS/SUM	16.741	SUBS/SUIM	17.02F	FCMOVGT
16.543	DIVS/SUM	16.742	MULS/SUIM	17.030	CVTQL
16.560	ADDT/SUM	16.743	DIVS/SUIM	17.130	CVTQL/V
16.561	SUBT/SUM	16.760	ADDT/SUIM	17.530	CVTQL/SV
16.562	MULT/SUM	16.761	SUBT/SUIM	18.0000	TRAPB
16.563	DIVT/SUM	16.762	MULT/SUIM	18.0400	EXCB
16.56C	CVTTS/SUM	16.763	DIVT/SUIM	18.4000	MB
16.56F	CVTTQ/SVM	16.76C	CVTTS/SUIM	18.4400	WMB
16.580	ADDS/SU	16.76F	CVTTQ/SVIM	18.8000	FETCH
16.581	SUBS/SU	16.77C	CVTQS/SUIM	18.A000	FETCH_M
16.582	MULS/SU	16.77E	CVTQT/SUIM	18.C000	RPCC
16.583	DIVS/SU	16.780	ADDS/SUI	18.E000	RC
16.5A0	ADDT/SU	16.781	SUBS/SUI	18.F000	RS
16.5A1	SUBT/SU	16.782	MULS/SUI	19	PAL19
16.5A2	MULT/SU	16.783	DIVS/SUI	1A.0	JMP
16.5A3	DIVT/SU	16.7A0	ADDT/SUI	1A.1	JSR
16.5A4	CMPTUN/SU	16.7A1	SUBT/SUI	1A.2	RET
16.5A5	CMPTEQ/SU	16.7A2	MULT/SUI	1A.3	JSR_COROUTINE
16.5A6	CMPTLT/SU	16.7A3	DIVT/SUI	1B	PAL1B
16.5A7	CMPTLE/SU	16.7AC	CVTTS/SUI	1C	OPC1C
16.5AC	CVTTS/SU	16.7AF	CVTTQ/SVI	1D	PAL1D

Table C–7 (Cont.): Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
1E	PAL1E	2A	LDL_L	36	FBGE
1F	PAL1F	2B	LDQ_L	37	FBGT
20	LDF	2C	STL	38	BLBC
21	LDG	2D	STQ	39	BEQ
22	LDS	2E	STL_C	3A	BLT
23	LDT	2F	STQ_C	3B	BLE
24	STF	30	BR	3C	BLBS
25	STG	31	FBEQ	3D	BNE
26	STS	32	FBLT	3E	BGE
27	STT	33	FBLE	3F	BGT
28	LDL	34	BSR		
29	LDQ	35	FBNE		

C.6 OpenVMS AXP PALcode Instruction Summary

Table C–8: OpenVMS AXP 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
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 interlocked resident
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–9: OpenVMS AXP 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

C.7 DEC OSF/1 PALcode Instruction Summary

Table C–10: DEC OSF/1 Unprivileged PALcode Instructions

Mnemonic	Opcode	Description
bpt	00.0080	Breakpoint trap
bugchk	00.0081	Bugcheck
callsys	00.0083	System call
gentrap	00.00AA	Generate software trap
imb	00.0086	I-stream memory barrier
rdunique	00.009E	Read unique value
wrunique	00.009F	Write unique value

Table C–11: DEC OSF/1 Privileged PALcode Instructions

Mnemonic	Opcode	Description
cflush	00.0001	Cache flush
cserve	00.0009	Console service
draina	00.0003	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

C.8 Windows NT AXP Instruction Summary

Table C–12: Windows NT AXP 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–13: Windows NT AXP Privileged PALcode instructions

Mnemonic	Opcode	Description
csir	00.000D	Clear software interrupt request
di	00.0008	Disable interrupts
draina	00.0002	Drain aborts
dtbis	00.0016	Data translation buffer invalidate single
ei	00.0009	Enable interrupts
halt	00.0000	Trap to illegal instruction
initpal	00.0004	Initialize the PALcode
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
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
tbis	00.0015	Translation buffer invalidate single
tbisasn	00.0017	Translation buffer invalidate single for ASN
wrentry	00.0005	Write system entry
wrmces	00.0013	Write machine check error summary
wrperfmon	00.0020	Write performance monitoring values

Opcodes 00.0038_{16} through $00.003F_{16}$ are reserved for processor implementation-specific PALcode instructions. All other opcodes are reserved for Digital's use.

C.9 PALcode Opcodes in Numerical Order

Table C–14: PALcode Opcodes in Numerical Order

Opcode_{16}	Opcode_{10}	OpenVMS AXP	DEC OSF/1	Windows NT AXP
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	—	wrperfmon
00.0021	00.0033	MTPR_SSP	—	—
00.0022	00.0034	MFPR_USP	—	—
00.0023	00.0035	MTPR_USP	—	—
00.0024	00.0036	MTPR_TBISD	—	—
00.0025	00.0037	MTPR_TBISI	—	—
00.0026	00.0038	MFPR_ASTEN	—	—
00.0027	00.0039	MFPR_ASTSR	—	—
00.0029	00.0040	MFPR_VPTB	—	—
00.002A	00.0041	MTPR_VPTB	—	—
00.002B	00.0042	MTPR_PERFMON	wrfen	—
00.002E	00.0043	MTPR_DATFX	—	—

Table C–14 (Cont.): PALcode Opcodes in Numerical Order

Opcode$_{16}$	Opcode$_{10}$	OpenVMS AXP	DEC OSF/1	Windows NT AXP
00.002D	00.0045	—	wrvptptr	—
00.0030	00.0048	—	swpctx	
00.0031	00.0049	—	wrval	
00.0032	00.0050	—	rdval	—
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	—
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	—
00.003F	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	—	—
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	—

Table C–14 (Cont.): PALcode Opcodes in Numerical Order

Opcode$_{16}$	Opcode$_{10}$	OpenVMS AXP	DEC OSF/1	Windows NT AXP
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	—	—
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

C.10 Required PALcode Function Codes

The opcodes listed in Table C–15 are required for all Alpha AXP 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–15: Required PALcode Function Codes

Mnemonic	Type	Function Code
DRAINA	Privileged	00.0002
HALT	Privileged	00.0000
IMB	Unprivileged	00.0086

C.11 Opcodes Reserved to PALcode

The opcodes listed in Table C–16 are reserved for use in implementing PALcode.

Table C–16: Opcodes Reserved for PALcode

Mnemonic		Mnemonic		Mnemonic	
PAL19	19	PAL1B	1B	PAL1D	1D
PAL1E	1E	PAL1F	1F		

C.12 Opcodes Reserved to Digital

The opcodes listed in Table C–17 are reserved to Digital.

Table C–17: Opcodes Reserved for Digital

Mnemonic		Mnemonic		Mnemonic	
OPC01	01	OPC02	02	OPC03	03
OPC04	04	OPC05	05	OPC06	06
OPC07	07	OPC0A	0A	OPC0C	0C
OPC0D	0D	OPC0E	0E	OPC14	14
OPC1C	1C				

C.13 Unused Function Code Behavior

Unused function codes for all opcodes assigned (not reserved) in the Version 5 Alpha AXP 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 AXP 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.14 ASCII Character Set

This section contains the 7-bit ASCII character set and the corresponding hexadecimal value for each character.

Table C–18: 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

Index

A

D

E

F

G

H

I

P

Q

R

S

T

U

V

W

X

Z

Technical Support and Ordering Information

Technical Support

If you need technical support or help deciding which literature best meets your needs, call the Semiconductor Information Line:

United States and Canada	**1–800–332–2717**
TTY (United States only)	**1–800–332–2515**
Outside North America	**+1–508–568–6868**

Ordering Digital Semiconductor Products

To order Digital Semiconductor products, contact your local Digital sales office. Your sales representative may be able to help you take advantage of discounts and volume pricing.

Ordering Associated Literature

For a list of available Digital Semiconductor literature, contact the Semiconductor Information Line.

Electronic Orders

To place an order through your account at the Electronic Store, dial 1–800–234–1998, using a modem set to 2400- or 9600-baud. You must use a VT terminal or terminal emulator set at 8 bits, no parity. If you need assistance using the Electronic Store, call 1–800–DIGITAL (1–800–344–4825) and ask for an Electronic Store specialist.

Telephone and Direct Mail Orders

From	Call	Write
U.S.A.	DECdirect Phone: 1–800–DIGITAL (1–800–344–4825) FAX: 603–884–5597	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Puerto Rico	Phone: 809–781–0505 FAX: 809–749–8377	Digital Equipment Corporation 3 Digital Plaza, 1st Street Suite 200 Metro Office Park San Juan, Puerto Rico 00920
Canada	Phone: 1–800–267–6215 FAX: 613–592–1946	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: DECdirect Sales
International	—	Local Digital subsidiary or approved distributor
Internal orders	DTN 264–3030 603–884–3030 FAX: 603–884–9690	U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063–1260