digital

Alpha System Reference Manual

Version 5

DIGITAL RESTRICTED DISTRIBUTION

Alpha System Reference Manual

Version 5

This document describes the Alpha architecture.

Digital Equipment Corporation
Maynard, Massachusetts

May 1992

This document was prepared using VAX DOCUMENT, Version 2.0.

Preface

The *Alpha System Reference Manual* is divided into 3 Parts, 4 appendixes, and an index.

Each part or section of a part describes a major portion of the Alpha architecture. Each contains its own Table of Contents. Additional sections will be incorporated as development proceeds on the architecture.

The *Alpha System Reference Manual* is under ECO control. ECOs are approved only by the Alpha-A committee.

The following table outlines the contents of the Alpha SRM:

Name	Symbol	Contents
Part One	(I)	Common Architecture This part describes the architecture that is common to and required by all implementations.
Part Two	(II) (III)	Specific Operating System PALcode Architecture This part contains *sections* that describe how the following operating systems relate to the Alpha architecture:

Section Name and Contents	Symbol
OpenVMS Alpha Software	(II)
DEC OSF/1 Alpha Software	(III)

Part Three	(IV)	Platforms This part describes an architected platform implementation.
Appendixes		Because information in the appendixes can be shared by more than one section, they are grouped together at the end of the manual.
Index		The index at the end of the manual is structured like a master index. Index entries are called out by the appropriate symbol, (I), (II), and so forth, associated with the corresponding part or section. Index entries for the appendixes are called out by appendix name and page number.

Common Architecture (I)

This part describes the common Alpha architecture and contains the following chapters:

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

Contents

Common Architecture (I)

Chapter 1 Introduction (I)

Chapter 2 Basic Architecture (I)

Chapter 3 Instruction Formats (I)

Chapter 4 Instruction Descriptions (I)

Chapter 5 System Architecture and Programming Implications (I)

Chapter 6 Common PALcode Architecture (I)

Chapter 7 Console Subsystem Overview (I)

Chapter 8 Input/Output (I)

Figures

Tables

Chapter 1
Introduction (I)

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

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

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

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

1.1 The Alpha Approach to RISC Architecture

Alpha Is a True 64-Bit Architecture

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

Alpha Is Designed for Very High-Speed Implementations

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

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

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

Alpha makes it easy to maintain binary compatibility across multiple implementations and easy to maintain full speed on multiple-issue implementations. For example, there are no implementation-specific pipeline timing hazards, no load-delay slots, and no branch-delay slots.

Alpha's Approach to Byte Manipulation

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

Alpha 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.
- Alpha's 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.
- Alpha's approach can make it easier to pipeline multiple byte operations.

Alpha's Approach to Arithmetic Traps

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

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

Alpha's Approach to Multiprocessor Shared Memory

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

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

Alpha Instructions Include Hints for Achieving Higher Speed

A number of Alpha instructions include hints for implementations, all aimed at achieving higher speed.

- Calculated jump instructions have a target hint that can allow much faster subroutine calls and returns.
- There are prefetching hints for the memory system that can allow much higher cache hit rates.

- There are granularity hints for the virtual-address mapping that can allow much more effective use of translation lookaside buffers for large contiguous structures.

PALcode—Alpha's Very Flexible Privileged Software Library

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

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

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

One version of PALcode lets Alpha implementations run the full OpenVMS operating system by mirroring many of the OpenVMS VAX features. The OpenVMS PALcode instructions let Alpha run OpenVMS with little more hardware than that found on a conventional RISC machine: the PAL mode bit itself, plus 4 extra protection bits in each Translation Buffer entry.

Another version of PALcode lets Alpha implementations run the OSF/1 operating system by mirroring many of the RISC ULTRIX features. Other versions of PALcode can be developed for real-time, teaching, and other applications.

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

Alpha and Programming Languages

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

- Alpha does not contain a subroutine call instruction that moves a register window by a fixed amount. Thus, Alpha is a good match for programming languages with many parameters and programming languages with no parameters.
- Alpha does not contain a global integer overflow enable bit. Such a bit would need to be changed at every subroutine boundary when a FORTRAN program calls a C program.

1.2 Data Format Overview

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

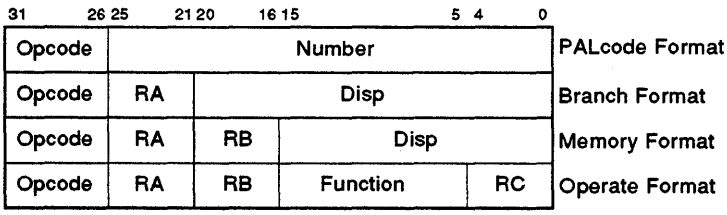
- All operations are done between 64-bit registers.
- Memory is accessed via 64-bit virtual little-endian byte addresses.
- There are 32 integer registers and 32 floating-point registers.
- Longword (32-bit) and quadword (64-bit) integers are supported.

- Four floating-point data types are supported:
  - — VAX F_floating (32-bit)
  - — VAX G_floating (64-bit)
  - — IEEE single (32-bit)
  - — IEEE double (64-bit)

1.3 Instruction Format Overview

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

Figure 1–1: Instruction Format Overview



- **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 Ra 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 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 architecture has no 32/64 mode bit.

Integer Operate Instructions

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

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

There is no integer divide instruction.

The Alpha architecture also supports the following additional operations:

- Scaled add/subtract instructions for quick subscript calculation
- 128-bit multiply for division by a constant, and multiprecision arithmetic
- Conditional move instructions for avoiding branch instructions

- An extensive set of in-register byte and word manipulation instructions

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

Floating-Point Operate Instructions

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

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

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

1.5 Instruction Set Characteristics

Alpha instruction set characteristics are as follows:

- All instructions are 32 bits long and have a regular format.
- There are 32 integer registers (R0 through R31), each 64 bits wide. R31 reads as zero, and writes to R31 are ignored.
- 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 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 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.

NOTE

\A note on the manual format: At certain points in the manual, comments on why certain decisions were made, unresolved issues, etc., are between a pair of backslashes. These comments provide additional clarification and will be removed from externally distributed editions.\

1.6.12 Macro Code Example Conventions

All instructions in macro code examples are either listed in Chapter 4 or *OpenVMS Section, Chapter 2*, or are stylized code forms found in *Appendix A*.

1.7 \Revision History

Revision 5.0, May 12, 1992

1. VMS –> OpenVMS
2. Converted to SDML
3. Removed reference to EVAX

Revision 4.0, March 29, 1991

1. Typos
2. Correct security holes text
3. Upgrade UNPREDICTABLE definition
4. Add Implementation Dependent definition
5. Add new section, Section 1.6.12, Macro Code Example Conventions

Revision 3.0, March 2, 1990

1. Strengthen UNPREDICTABLE definition
2. Add UNALIGNED definition
3. Add Security Hole definition

Revision 2.0, October 4, 1989

1. Change the read as zero, write ignored registers to R31 and F31
2. Update instruction Set Characteristics for new insert and merge byte instructions

Revision 1.0, May 23, 1989

1. Change MBZ and SBZ definitions

Revision 0.0, March 15, 1988

1. Initial version

Chapter 2

Basic Architecture (I)

2.1 Addressing

The basic addressable unit in Alpha 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.

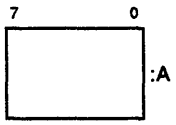
2.2 Data Types

Following are descriptions of the Alpha architecture data types.

2.2.1 Byte

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

Figure 2–1: Byte Format

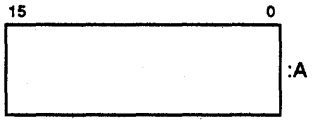


A byte is specified by its address A. A byte is an 8-bit value. The byte is only supported in Alpha by the extract, mask, insert, and zap instructions.

2.2.2 Word

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

Figure 2–2: Word Format



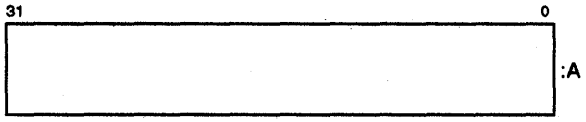
A word is specified by its address, the address of the byte containing bit 0.

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

2.2.3 Longword

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

Figure 2–3: Longword Format



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

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

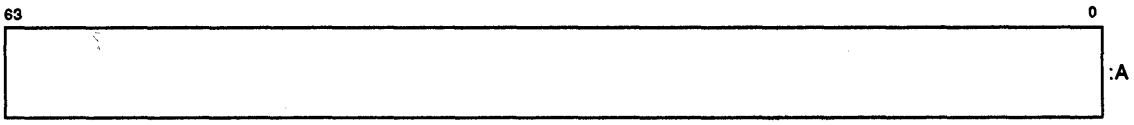
NOTE

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

2.2.4 Quadword

A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 63, as shown in Figure 2–4.

Figure 2–4: Quadword Format



A quadword is specified by its address A, the address of the byte containing bit 0. A quadword is a 64-bit value. When interpreted arithmetically, a quadword is either a two's-complement integer with bits of increasing significance from 0 through 62 and bit 63 as the sign bit, or an unsigned integer with bits of increasing significance from 0 through 63.

NOTE

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

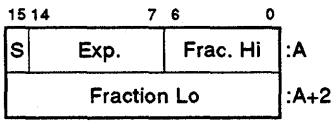
2.2.5 VAX Floating-Point Formats

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

2.2.5.1 F_floating

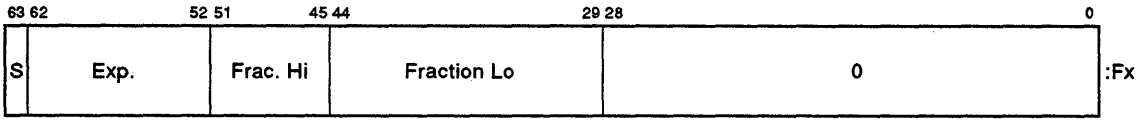
An F_floating datum is 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2–5.

Figure 2–5: F_floating Datum



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

Figure 2–6: F_floating Register Format



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

Table 2–1: F_floating Load Exponent Mapping

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..1.7*10**38. The precision of an F_floating datum is approximately one part in 2**23, typically 7 decimal digits.

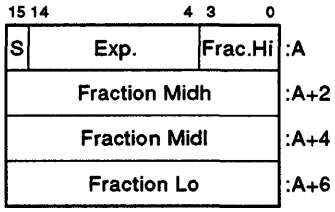
NOTE

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

2.2.5.2 G_floating

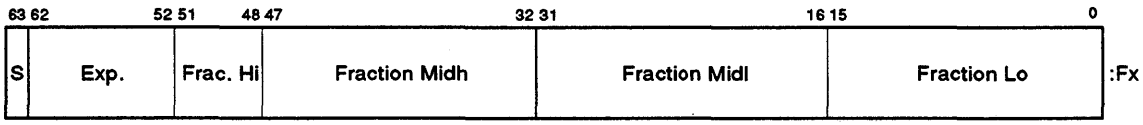
A G_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–7.

Figure 2–7: G_floating Datum



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

Figure 2–8: G_floating Format



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

If the result of a floating-point instruction has a value of zero, the instruction always produces a datum with a sign bit of 0, an exponent of 0, and all fraction bits of 0. Exponent values of 1..2047 indicate true binary exponents of

–1023..1023. An exponent value of 0, together with a sign bit of 1, is taken as a reserved operand. Floating-point instructions processing a reserved operand take a user-visible arithmetic exception. The value of a G_floating datum is in the approximate range 0.56*10**–308..0.9*10**308. The precision of a G_floating datum is approximately one part in 2**52, typically 15 decimal digits.

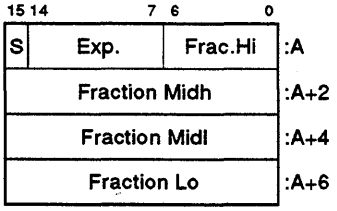
NOTE

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

2.2.5.3 D_floating

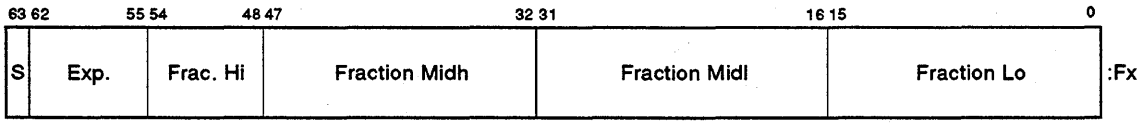
A D_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–9.

Figure 2–9: D_floating Datum



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

Figure 2–10: D_floating Register Format



The reordering of bits required for a D_floating load or store 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.

NOTE

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.

NOTE

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

2.2.6 IEEE Floating-Point Formats

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

1. P—the number of fraction bits
2. 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

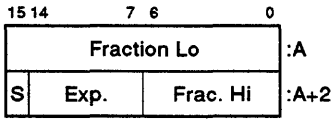
NaN is an acronym for Not-a-Number. A NaN is an IEEE floating-point bit pattern that represents something other than a number. NaNs come in two forms: Signaling NaNs and Quiet NaNs. Signaling NaNs are used to provide values for uninitialized variables and for arithmetic enhancements. Quiet NaNs provide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

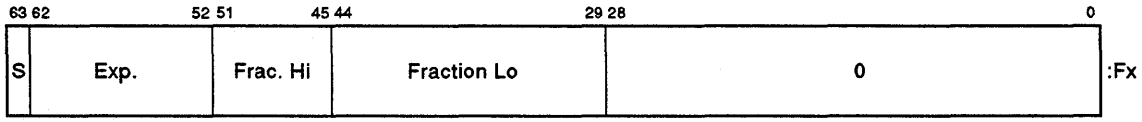
An IEEE single-precision, or S_floating, datum occupies 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2–11.

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

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.

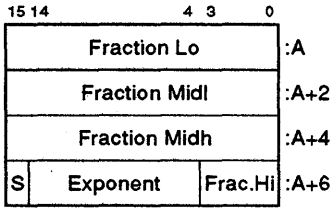
NOTE

Alpha 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

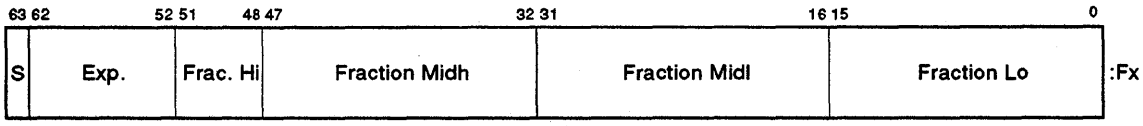
An IEEE double-precision, or T_floating, datum occupies 8 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–13.

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.

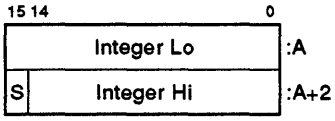
NOTE

Alpha 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.)

2.2.7 Longword Integer Format in Floating-Point Unit

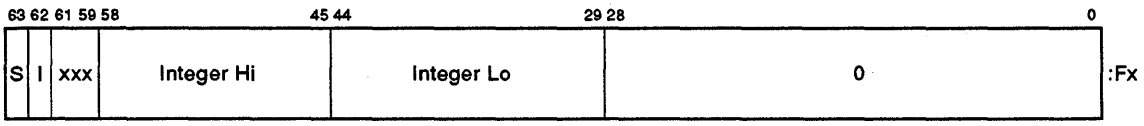
A longword integer operand occupies 32 bits in memory, arranged as shown in Figure 2–15.

Figure 2–15: Longword Integer Datum



A longword integer operand occupies 64 bits in a floating register, arranged as shown in Figure 2–16.

Figure 2–16: 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 2–16 is part of the Integer Hi field in Figure 2–15 and represents the high-order bit of that field. Bits <58:45> of Figure 2–16 are the remaining bits of the Integer Hi field of Figure 2–15.

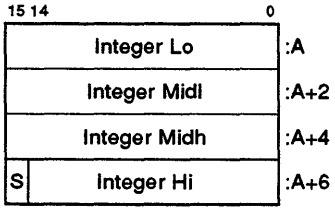
NOTE

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

2.2.8 Quadword Integer Format in Floating-Point Unit

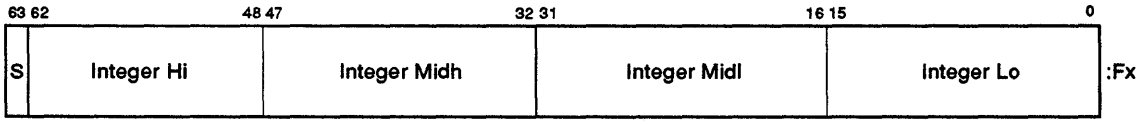
A quadword integer operand occupies 64 bits in memory, arranged as shown in Figure 2–17.

Figure 2–17: Quadword Integer Datum



A quadword integer operand occupies 64 bits in a floating register, arranged as shown in Figure 2–18.

Figure 2–18: Quadword Integer Floating-Register Format



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

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

NOTE

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

2.2.9 Data Types with No Hardware Support

The following VAX data types are not directly supported in Alpha hardware. \ See the *DEC STD 032: VAX Architecture Standard* for detailed information on these data types. \

- 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 \Revision History

Revision 5.0, May 12, 1992

1. Converted to SDML

Revision 4.0, March 29, 1991

1. D_floating point support removed
2. Typos
3. Word definition made homologous to longword, quadword
4. Specify no checking on S_floating load, and T_floating load
5. Removed S_floating Format illustration and text
6. Clarified what is meant by a Vax floating point instruction

Revision 3.0, March 2, 1990

1. Cosmetic change to floating-point pictures

Revision 2.0, October 4, 1989

1. No change

Revision 1.0, May 23, 1989

1. Change minimum virtual address size to 40 bits
2. Change Floating-point register format
3. Remove alignment warning on word data type

Revision 0.0, March 15, 1989

1. Initial version

\

Chapter 3

Instruction Formats (I)

3.1 Alpha Registers

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

3.1.1 Program Counter

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

3.1.2 Integer Registers

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

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

For all cases except the Unconditional Branch and Jump instructions, results of an instruction that specifies R31 as a destination operand are discarded. Also, it is UNPREDICTABLE whether the other destination operands (implicit and explicit) are changed by the instruction. It is implementation dependent to what extent the instruction is actually executed once it has been fetched. 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 4.7.2 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 Optional Registers

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

3.1.5.1 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.

3.1.5.2 VAX Compatibility Register

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

3.2 Notation

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

3.2.1 Operand Notation

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

Table 3–1: Operand Notation

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

Table 3–2: Operand Value Notation

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

Table 3–3: Expression Operand Notation

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

3.2.2 Instruction Operand Notation

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

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

<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 PAL function field of the instruction.
Ra	An integer register operand in the Ra field of the instruction.
Rb	An integer register operand in the Rb field of the instruction.
#b	An integer literal operand in the Rb field of the instruction.
Rc	An integer register operand in the Rc field of the instruction.
Fa	A floating-point register operand in the Ra field of the instruction.
Fb	A floating-point register operand in the Rb field of the instruction.
Fc	A floating-point register operand in the Rc field of the instruction.

<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.

Access Type	Meaning
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

The operators shown in Table 3–4 are used:

Table 3–4: Operators

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

Table 3–4 (Cont.): Operators

Operator	Meaning
x<m>	M'th bit of x
ACCESS(x,y)	Accessibility of the location whose address is x using the access mode y. Returns a Boolean value TRUE if the address is accessible, else FALSE.
AND	Logical product
ARITH_RIGHT_SHIFT(x,y)	Arithmetic right shift of first operand by the second operand. Y is an unsigned shift value. Bit 63, the sign bit, is copied into vacated bit positions and shifted out bits are discarded.
BYTE_ZAP(x,y)	X is a quadword, y is an 8-bit vector in which each bit corresponds to a byte of the result. The y bit to x byte correspondence is y<n> ↔ x<8n+7:8n>. This correspondence also exists between y and the result. For each bit of y from n = 0 to 7, if y <n> is 0 then byte <n> of x is copied to byte <n> of result, and if y <n> is 1 then byte <n> of result is forced to all zeros.

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

Table 3–4 (Cont.): Operators

Operator	Meaning
NOT	Logical (ones) complement
OR	Logical sum
x MOD y	x modulo y
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

MINU(x,y)	Returns the smaller of x and y, with x and y interpreted as unsigned integers
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`
RIGHT_SHIFT(x,y)	Logical right shift of first operand by the second operand. Y is an unsigned shift value. Zeros are moved into vacated bit positions, and shifted out bits are discarded.
SEXT(x)	X is sign-extended to the required size.
STORE_CONDITIONAL	If the lock_flag is set, then do the indicated store and clear the lock_flag.

Table 3–4 (Cont.): Operators

Operator	Meaning
TEST(x,cond)	The contents of register x are tested for branch condition (cond) true. TEST returns a Boolean value TRUE if x bears the specified relation to 0, else FALSE is returned. Integer and floating test conditions are drawn from the preceding list of relational operators.
XOR	Logical difference
ZEXT(x)	X is zero-extended to the required size.

3.2.4 Notation Conventions

The following conventions are used:

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

3.3 Instruction Formats

There are five basic Alpha instruction formats:

- Memory
- Branch
- Operate
- Floating-point Operate
- PALcode

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

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

SOFTWARE NOTE

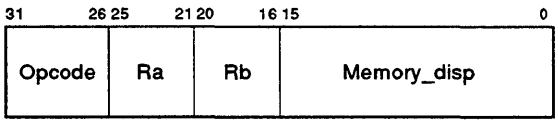
There are several instructions, each formatted as a memory instruction, that do not use the Ra and/or Rb fields. These instructions are: Memory Barrier, Fetch,

Fetch_M, Read Process Cycle Counter, Read and Clear, Read and Set, and Trap Barrier.

3.3.1 Memory Instruction Format

The Memory format is used to transfer data between registers and memory, to load an effective address, and for subroutine jumps. It has the format shown in Figure 3–1.

Figure 3–1: Memory Instruction Format



A Memory format instruction contains a 6-bit opcode field, two 5-bit register address fields, Ra and Rb, and a 16-bit signed displacement field.

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

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

```
va ← {Rbv + SEXT(Memory_disp)}
```

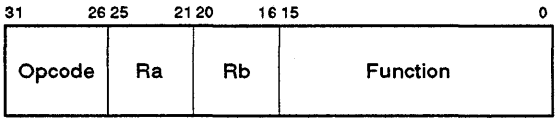
For LDAH the virtual address (va) is computed as follows:

```
va ← {Rbv + SEXT(Memory_disp*65536)}
```

3.3.1.1 Memory Format Instructions with a Function Code

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

Figure 3–2: Memory Instruction with Function Code Format



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

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

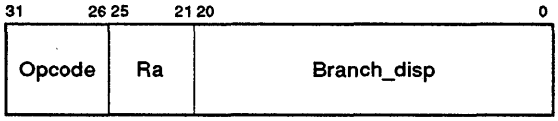
3.3.1.2 Memory Format Jump Instructions

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

3.3.2 Branch Instruction Format

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

Figure 3–3: Branch Instruction Format



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

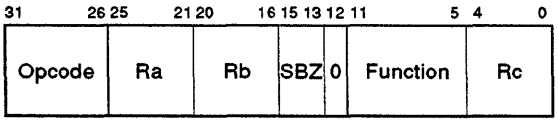
The displacement is treated as a longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits and added to the updated PC to form the target virtual address. Overflow is ignored in this calculation. The target virtual address (va) is computed as follows:

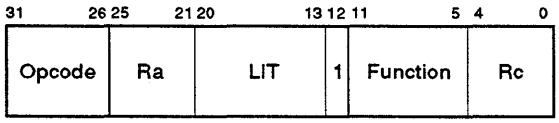
```
va ← PC + {4*SEXT(Branch_disp)}
```

3.3.3 Operate Instruction Format

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

Figure 3–4: Operate Instruction Format





An Operate format instruction contains a 6-bit opcode field and a 7-bit function field. Unused function encodings produce UNPREDICTABLE but not UNDEFINED results; they are not security holes.

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

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

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

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

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

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

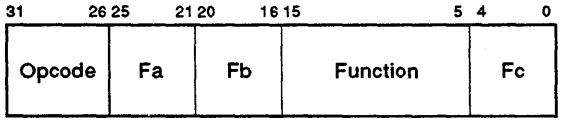
```
IF inst<12> EQ 1 THEN
    Rbv ← ZEXT(inst<20:13>)
ELSE
    IF inst<20:16> EQ 31 THEN
        Rbv ← 0
    ELSE
        Rbv ← Rb
    END
END
```

The Rc field specifies a destination operand.

3.3.4 Floating-Point Operate Instruction Format

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

Figure 3–5: Floating-Point Operate Instruction Format



A Floating-point Operate format instruction contains a 6-bit opcode field and an 11-bit function field. Unused function encodings produce UNPREDICTABLE results, as defined in Section 1.6.3.

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

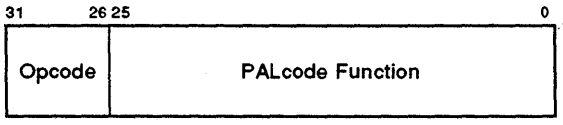
Floating-point Convert instructions use a subset of the Floating-point Operate format and perform register-to-register conversion operations. The Fb operand specifies the source; the Fa field must be F31.

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.

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.

3.4 \Revision History

Revision 5.0, May 12, 1992

1. Removed references to SP and PS
2. Added unsigned multiplication operator
3. Added description of Fa, Fb registers if unused
4. Converted to SDML
5. Added Memory Format with Function Code section
6. Moved Instruction Operand section from Chapter 4
7. Edited description of R31
8. Separated operand notation from operand value notation and simplified language
9. Added comment and note to section 3.3 which specifies value assigned to unused register fields of instructions

Revision 4.0, March 29, 1991

1. Typos
2. Upgrade description of R30 and implicit stack behavior of HW/PALcode
3. Upgrade definition of byte_zap, access, left_shift, and right_shift operators
4. Add definition of single bit field select operator, <n>
5. Rename arith_shift operator to arith_right_shift and upgrade definition
6. Make test a dyadic operator with explicit condition argument
7. Define the CASE pseudocode construct
8. Include Processor Status register in description of Alpha registers
9. Add definitions of priority_encode and exponentiation (**) operators
10. Changed text describing R30
11. Changed two relational operator mnemonics

Revision 3.0, March 2, 1990

1. Under registers, add lock registers, IPRs, and optional registers
2. Define DIV, BYTE_ZAP, and PHYSICAL_ADDRESS; delete BYTE_SEL
3. Delete reference to R28

Revision 2.0, October 4, 1989

1. Add comment to section on PC that PC is not an Integer Register
2. Add comment that SP is R30

3. Change description of L field in operate Instruction format

Revision 1.0, May 23, 1989

1. Remove Rb reading as PC for Rb eq 0
2. Fix error in which bit is literal enable bit for operate format
3. Add Floating-point Operate format

Revision 0.0, March 15, 1989

1. Initial version

\

Chapter 4
Instruction Descriptions (I)

4.1 Instruction Set Overview

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

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

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

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

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

4.1.2 Software Emulation Rules

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

Emulation of loads and stores of unaligned data and subsetted instructions need not be provided in privileged access modes. System software that supports special-

purpose dedicated applications need not provide emulation in User mode if emulation is not needed for correct execution of the special-purpose applications.

4.1.3 Opcode Qualifiers

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

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

Table 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)}

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. 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)}

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. 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 8 (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 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 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.mq,disp.ab(Rb.ab)          !Memory format
```

Operation:

```
va ←  {Rbv + SEXT(disp)}

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. If the lock_flag is set, the Ra operand is written to memory at this address. (See the LDx_L description for conditions that clear the lock_flag.) The lock_flag is returned in RA and then set to a zero.

Notes:

- Software will not emulate unaligned STx_C instructions.
- Each implementation must do the test and store atomically, so that if two processors execute store conditionals within the same lock range, exactly one of the stores succeeds.

- The following sequence should not be used:

```
try_again: LDQ_L   R1,x
           <modify R1>
           STQ_C   R1,x
           BEQ     R1, try_again
      .
      .
      .
```

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

SOFTWARE NOTE

The address specified by a STx_C instruction need not match that given in a preceding LDx_L. Specifying unmatched addresses for those instructions requires an MB in between to guarantee ordering.

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

Operation:

```
va ←   {Rbv + SEXT(disp)}
(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. 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 provides integer conditional branch, unconditional branch, branch to subroutine, and jump instructions. The PC used in these instructions is the updated PC, as described in Section 3.1.1.

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

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

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

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

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

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

Table 4–3: Control Instructions Summary

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.rq,Rb.rq,Rc.wq          !Operate format
ADDL    Ra.rq,#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.rq,Rb.rq,Rc.wq	!Operate format
SxADDL	Ra.rq,#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.rq,Rb.rq,Rc.wq          !Operate format
MULL    Ra.Rq,#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.rq,Rb.rq,Rc.wq          !Operate format
SUBL    Ra.rq,#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.rq,Rb.rq,Rc.wq	!Operate format
SxSUBL	Ra.rq,#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.rb,#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 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
     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.

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
SLL     R1,#32,R1       ; R1 = DCBA 0000
SRA     R1,#32,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(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
SLL     R1,#48,R1       ; R1 = BA00 0000
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
```

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
    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 Rb<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
    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 Rb<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, and the value to be stored from R5 is hgfe dcba.

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 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 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 registers.

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

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

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

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

An implementation including 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.

4.7.2 Definitions

The following definitions apply to Alpha floating-point support.

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.

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.

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.

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

Alpha finite number
A floating-point number with a definite, in-range value. Specifically, all numbers in the inclusive ranges –MAX..–MIN, zero, +MIN..+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.

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

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

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.

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.

4.7.3 Encodings

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

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

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

Data Format	MIN	MAX
F_floating	2**–127 * 0.5 (0.294e–38)	2**127 * (1.0 – 2**–24) (1.70e38)
G_floating	2**–1023 * 0.5 (0.56e–308)	2**1023 * (1.0 – 2**–53) (0.899e308)
S_floating	2**–126 * 1.0 (1.175e–38)	2**127 * (2.0 – 2**–23) (3.40e38)

Data Format	MIN	MAX
T_floating	2**–1022 * 1.0 (2.225e–308)	2**1023 * (2.0 – 2**–52) (1.798e308)

4.7.4 Floating-Point Rounding Modes

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

VAX Rounding Modes

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

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

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

IEEE Rounding Modes

For IEEE floating-point operations, four rounding modes are provided: normal rounding (unbiased round to nearest), rounding toward minus infinity, round toward zero, and rounding toward plus infinity. The first three can be specified in the 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 IEEE arithmetic does rounding before detecting overflow/underflow.

Normal IEEE rounding maps the true result to the nearest of two representable results, with true results exactly halfway between mapped to the one whose fraction ends in 0 (sometimes called unbiased rounding to even); maps true results $\geq$ 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 $\leq$ +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 $\geq$ –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

4.7.5 Floating-Point Trapping Modes

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

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

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

VAX Trapping Modes

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

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

IEEE Trapping Modes

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

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

The modes and instruction notation are:

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
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

4.7.5.1 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 following conditions must hold:

1. On entry to the trap shadow, if any Alpha register or memory location contains a value that is used as an operand value by some instruction in the trap shadow (live on entry), then no instruction in the trap shadow may modify the register or memory location.
2. 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.
3. Within the trap shadow, no register may be used more than once as a destination register.
4. The trap shadow may not include any branch instructions.
5. Each floating instruction to be completed must be so marked, by specifying the /S software completion modifier.

The first condition allows 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.

The second condition prevents memory accesses at unpredictable addresses.

The remaining conditions make it possible for a software trap handler to find the trigger instruction via a linear scan backwards from the trap PC.

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 the trap 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.

4.7.5.2 Invalid Operation Arithmetic Trap

An invalid operation arithmetic trap is signaled if any operand of a floating arithmetic-operate instruction is non-finite. (CMPTxy is an exception to the rule and operates normally with plus and minus infinity and does not trap in this case.) This trap is always enabled. If this 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, 0/0, x REM 0, and conversions to integer that take an integer overflow trap.)

4.7.5.3 Division by Zero 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. This trap is always enabled. If this trap occurs, an UNPREDICTABLE value is stored in the result register.

4.7.5.4 Overflow Arithmetic Trap

An overflow arithmetic trap is signaled if the rounded result exceeds in magnitude the largest finite number of the destination format. This trap is always enabled. If this trap occurs, an UNPREDICTABLE value is stored in the result register.

4.7.5.5 Underflow 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.

4.7.5.6 Inexact Result 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.

4.7.5.7 Integer Overflow 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 Floating-Point Single-Precision Operations

Single-precision values (F_floating or S_floating) are stored in the floating 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 registers are stored in bits <63:62,58:29>, with bits <61:59> ignored and zeros in bits <28:0>.

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 code bits <7:6> = 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.

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

the exceptional condition was encountered by an instruction will still be recorded in the FPCR.

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

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

The 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

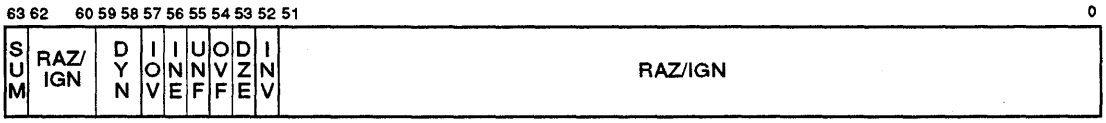


Table 4–8: Floating-Point Control Register (FPCR) Bit Descriptions

Bit	Description
63	Summary Bit (SUM). Records bitwise OR of FPCR exception bits. Equal to (FPCR[57] | FPCR[56] | FPCR[55] | FPCR[54] | FPCR[53] | FPCR[52]).
62–60	Reserved. Read As Zero; Ignored when written.
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

57	Integer Overflow (IOV). An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision.
56	Inexact Result (INE). A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.

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

Bit	Description
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.
51–0	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 4.1.1.1). 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

As noted in Section 4.1.1.1, support for FPCR is required on a system that supports the OpenVMS Alpha operating system even if that system does not support floating-point.

4.7.7.1 Accessing the FPCR

Because Alpha floating-point hardware can overlap the execution of a number of floating-point instructions, accessing the FPCR must be synchronized with other floating-point instructions. A TRAPB 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 a TRAPB followed by an MT_FPCR followed by another TRAPB ensures that only floating-point instructions issued after the second TRAPB are affected by and affect the new value of the FPCR. Issuing a TRAPB followed by an MF_FPCR followed by another TRAPB ensures that the value read from the FPCR only records the exception information for floating-point instructions issued prior to the first TRAPB.

Consider the following example:

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

Without the first TRAPB, 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 TRAPB, 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.

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> = 11 during program activation. Using this default, interval arithmetic code can switch from plus to minus infinity rounding with no penalty in performance by using /M and /D qualifiers.

Program activation should clear all other fields of the FPCR.

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 NOTE

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

4.7.8 IEEE Standard

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

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 4–9.

Table 4–9: 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)}
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 8-bit memory-format exponent is 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. 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)}
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 8-bit memory-format exponent is 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. The source operand is fetched from memory, is zero-extended in the low-order longword, and then written to register Fa.

Notes:

- 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)}
(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. 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)}
(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. The bits of the source operand are fetched from register Fa, the bits are reordered to conform to S_floating memory format, and the result is then written to memory. Bits <61:59> and <28:0> of Fa are ignored. No checking is done.

4.8.8 Store T_floating

Format:

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

Operation:

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

Exceptions:

Access Violation
Fault on Write
Alignment
Translation Not Valid

Instruction mnemonics:

STT Store T_floating (Store Quadword Integer)

Qualifiers:

None

Description:

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

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

4.9 Branch Format Floating-Point Instructions

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

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

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

The floating-point branch operations are summarized in Table 4–10.

Table 4–10: 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 4–11.

Table 4–11: 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 4–11 (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
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
SUBF	Subtract F_floating	VAX

Table 4–11 (Cont.): Floating-Point Operate Instructions Summary

Mnemonic	Operation	Subset
Arithmetic Operations		
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)
	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.

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.

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 4.7.5 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.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.5 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.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones and a non-zero fraction (IEEE NaNs). The contents of Fc are UNPREDICTABLE if this occurs.

Comparisons with plus and minus infinity execute normally and do not take an invalid operation trap. \ This was added to support fast path selection through infinity testing in scientific codes.\

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.

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 4.7.5 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.

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.

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 4.7.5 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.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.5 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.

See Section 4.7.5 for details of the stored result on inexact result.

4.10.14 Convert IEEE Floating to IEEE 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 floating operand in register Fb is converted to the specified alternate floating format and written to register Fc.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.5 for details of the stored result on overflow, underflow, or inexact result.

Notes:

- No conversion from S_floating to T_floating is required, since S_floating values are always stored in registers as equivalent T_floating values.

4.10.15 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 4.7.5 for details of the stored result on overflow or underflow.

4.10.16 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.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities 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 4.7.5 for details of the stored result on overflow, underflow, or inexact result.

4.10.17 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 4.7.5 for details of the stored result on overflow or underflow.

4.10.18 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 Eenable (/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.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.5 for details of the stored result on overflow, underflow, or inexact result.

4.10.19 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 4.7.5 for details of the stored result on overflow or underflow.

4.10.20 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.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.5 for details of the stored result on overflow, underflow, or inexact result.

4.11 Miscellaneous Instructions

Alpha provides the miscellaneous instructions shown in Table 4–12.

Table 4–12: Miscellaneous Instructions Summary

Mnemonic	Operation
CALL_PAL	Call Privileged Architecture Library Routine
FETCH	Prefetch Data
FETCH_M	Prefetch Data, Modify Intent
MB	Memory Barrier
RPCC	Read Process Cycle Counter
TRAPB	Trap 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 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.

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

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.

4.11.3 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.

4.11.4 Read Process Cycle Counter

Format:

```
RPCC    Ra.wq                          !Memory format
```

Operation:

```
Ra ← {cycle counter}
```

Exceptions:

None

Instruction mnemonics:

RPCC Read Process Cycle Counter

Qualifiers:

None

Description:

Register Ra is written with the process cycle counter (PCC).

The low-order 32 bits of the process cycle counter is an unsigned 32-bit integer that 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 process cycle counter gets incremented per second, rounded to a 64-bit integer. 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). The high-order 32 bits of the process cycle counter are an offset that when added to the low-order 32 bits gives the cycle count for this process.

The process cycle counter is suitable for timing intervals on the order of nanoseconds and may be used for detailed performance characterization. It is required on all implementations. PCC is required for every processor, and each processor in a multiprocessor system has its own private, independent PCC.

\INTERNAL IMPLEMENTATION NOTE

An implementation-dependent mechanism must exist that, when enabled, causes the RPCC instruction always to return a zero in Ra. This mechanism must be usable by privileged system software. \

As an example, consider the following code that returns in R0 the current cycle count MOD 2**32.

```
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
```

4.11.5 Trap Barrier

Format:

```
TRAPB                                   !Memory format
```

Operation:

```
{Stall instruction issuing until all prior instructions are
guaranteed to complete without incurring 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 any instructions after the TRAPB are issued. For example, TRAPB should be used before changing an exception handler to ensure that all exceptions on previous instructions are processed in the current exception-handling environment.

4.12 VAX Compatibility Instructions

Alpha provides the instructions shown in Table 4–13 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.

NOTE

\They will be removed, and not emulated, after the first two full generations of Alpha implementations, that is, about 1995. \

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

Table 4–13: VAX Compatibility Instructions Summary

Mnemonic	Operation
RC	Read and Clear
RS	Read and Set

4.12.1 VAX Compatibility Instructions

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

RC	Read and Clear
RS	Read and Set

Qualifiers:

None

Description:

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

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

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

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

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

NOTE

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

4.13 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. added eco #41 to LDx_C and format style change
2. Changed DRAINT to TRPB
3. Converted to SDML
4. Modifed description of MULQ to spec. operands and result are signed
5. Removed FCMOV and CVTLQ from instructions that set FPCR bits
6. Changed byte mask for INSxx and MSKxx instructions to 16 bit value

Revision 4.0, March 29, 1991

1. Added Scaled Add and Subtract
2. Added FPCR register and accompanying text
3. Bits <13:0> of branch displacement field in RET and JSR_COROUTINE reserved to Digital software
4. Removed references to D_floating point
5. Clarified floating-point subset requirements and added OpenVMS requirements for FP regs and T_floating memory ops in implementation without floating-point support
6. Make TEST a dyadic operator with explicit condition argument
7. Fix ADDQ to allow literal as second operand, not first
8. Add format type to Arithmetic and Logical and shift Instructions
9. Rename operator ARITH_SHIFT to ARITH_RIGHT_SHIFT and upgrade description
10. Add description of how to derive upper 64 bits of product (using UMULH) to MULQ description
11. Add requirement that F_, D_, and G_floating operate Instructions materialize a true zero
12. Clarify expressions for MAX F_, D_, G_, S_, and T_ values
13. Reorder special values table in floating-point encodings section
14. Modify MB description to indicate that MB works only on instructions from issuing processor
15. Disambiguate between instances when floating disabled faults and illegal instruction traps are taken
16. Clarify that low order bits are returned on integer overflow arithmetic conversion traps

17. Add description to STx_C Instruction that clarifies implementation requirements for execution of STxC Instruction
18. Correct decimal value given for MIN T_floating
19. Impose uniform usage of CASE pseudocode construct
20. Insert spaces into long hex and binary values to improve legibility
21. Added optimized sign-extended byte load code fragment to code examples in Extract Byte Instruction description
22. Clarify use and significance of X+C notation in code examples for Extract Byte Instruction
23. Clarify note describing how a Read For Ownership cache coherency protocol can affect LDx_L/STx_C sequence
24. Change reference in Floating-Point Operate Format Instructions from 'floating-point arithmetic operations' to 'floating-point operate Instructions'
25. Rename RCC instruction to 'Read Process Cycle Counter' and modify definition
26. Changed values of displacement bits <13:0> in 'Jump To Subroutine' instruction to indicate that all values from 0010 to 1111_{16} are reserved to Digital
27. Removed text in Longword Add instruction that described carry detection
28. Specified overflow bits returned for Longword Mulutiply
29. Removed text in Longword Subtract instruction that described carry detection

Revision 3.0, March 2, 1990

1. Rename GOTO to BR, and JSRs to JMP, JSB, RET
2. Rename MSKxx to ZAPxx
3. Remove CVTFQ, and CMPFxx
4. Remove CVT float-to-longword; add CVTQL/LQ
5. Make non-canonical longword +-* well-defined
6. Rename memory-format JSR to BSR
7. Add VAX compatibility Instructions RC, RS
8. Add Fetch and Fetch_M
9. Add low bit set and clear cmoves
10. Remove Nudge
11. Add longword lock Instructions
12. Remove longword load address Instructions
13. Add quadword load address high

14. Rework the LDx/L description
15. Change EXTxx/INSxx back to V1.0 SRM EXTxx/INSxx/MRGxx
16. Change floating-point exception behavior back to V1.0 SRM behavior

Revision 2.0, October 4, 1989

1. Add TLE provided comment on emulation of Instructions
2. Change shift range from 0..64 to 0..63
3. Remove FASx, SWP, FREEZE, THAW Instructions
4. Add load lock and store conditional Instructions
5. Remove WAIT/WAITF Instructions
6. Change DRAIN to DRAINT and only drain for arithmetic traps
7. Add memory barrier and nudge Instructions
8. Rework Floating-point exceptions
9. Add cycle counter

Revision 1.0, May 23, 1989

1. Rework Floating-point to be unmoded
2. Remove subsetting of integer MUL
3. Remove integer DIV
4. Add Freeze and Thaw
5. Rename Lock/Unlock to SWP and FASx and remove long version of lock
6. Add conditional move
7. Add branch on low bit branches (BLBS/BLBC)
8. Add WAIT/WAITF Instructions

Revision 0.0, March 15, 1989

1. Initial Version

Chapter 5

System Architecture and Programming Implications (I)

5.1 Introduction

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

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

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

5.2 Physical Memory Behavior

Alpha physical memory space is divided into four regions, based on the two most significant, implemented, physical address bits. Each region's behavior can be described in terms of its coherency, granularity, width, and memory-like behavior.

5.2.1 Coherency of Memory Access

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

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

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

1. No caching

   No copies are kept of data in a region; all reads and writes access the actual data location (memory or I/O register).

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.

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 Behavior

A memory-like region 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 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 nonsoftware 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 5.6.1.9.)
2. Write-back caches must be able to detect a later write from another processor and invalidate or update the cache contents.
3. A processor must guarantee that a data store to a location followed by a data load from the same location 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 sequence:

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

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

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 5.6.1.9.) 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 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>
        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 MB 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 do a store (no STx_C), or
- Having no write in the spin loop, or
- Doing a write *only* if the shared variable actually changes state ($1 \rightarrow 1$ does not change state).

5.5.4 Ordering Considerations for Shared Data Structures

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

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

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

```
<acquire software lock>
MB (memory barrier #1)
<critical section -- read/write shared data>
MB (memory barrier #2)
<clear software lock>
```

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 reads or writes (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.

SOFTWARE NOTE

In the VAX architecture, many instructions provide non-interruptable read-modify-write sequences to memory variables. Most programmers never regard data sharing as an issue.

In the Alpha architecture, programmers must pay more attention to synchronizing access to shared data; for

example, to AST routines. In the VAX, a programmer can use an ADDL2 to update a variable that is shared between a "MAIN" routine and an AST routine, if running on a single processor. In the Alpha architecture, a programmer must deal with AST shared data by using multiprocessor shared data sequences.

5.6 Read/Write Ordering

This section 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 implementation may differ from that specified by the programmer.

For any two memory references A and B, either A must occur before B in all Alpha implementations, B must occur before A, or they are UNORDERED. In the last case, software cannot depend upon one occurring first: the order may vary from implementation to implementation, and even from run to run or moment to moment on a single implementation.

If two references 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 references on a single processor, then defines the (partial) ordering on this issue sequence that *all* Alpha implementations are required to maintain.

The individual issue sequences on multiple processors are merged into access sequences at each shared memory location. The discussion defines the (partial) ordering on the individual access sequences that *all* Alpha implementations are required to maintain.

The net result is that for any code that executes on multiple processors, one can determine which memory accesses are required to occur before others on *all* Alpha implementations and hence can write useful shared-variable software.

Software writers can force one reference to occur before another by inserting a memory barrier instruction (MB or IMB) between the references.

5.6.1 Alpha Shared Memory Model

An Alpha system consists of a collection of *processors* and shared coherent *memories* that are accessible by all processors. (There may also be unshared memories, but they are outside the scope of this section.)

NOTE

\ Unshared example: On the PMI, some physical addresses in I/O space access unshared processor-local CSRs.\

A *processor* is an Alpha CPU or an I/O device (or anything else that gets added).

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.

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 five 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. 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 two types collectively are called barriers.

During actual execution in an Alpha system, each processor has a time-ordered *issue sequence* of all the memory references 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 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.

All memory references are aligned quadwords. For the purpose of defining ordering, aligned longword references are modeled as quadword references to the containing aligned quadword.

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: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: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).

Issue order is thus a partial order imposed on the architecturally specified issue sequence. Implementations are free to do 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 an 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 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.

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:

Table 5–2: Location Access Order

1st↓/2nd→	Pi:I(x,b)	Pi:R(x,b)	Pi: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 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.)

5.6.1.6 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.

5.6.1.7 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 ⇐ v) or
(u ≪ w and w ⇐ v).

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

5.6.1.8 Definition of After

If u ⇐ v, then *v* is said to be AFTER *u*.

At most one of u ⇐ v and v ⇐ u is true.

5.6.1.9 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 ≪ V1, by the definition of storage
V2 reading 1 implies V2 ≪ U1, by the definition of storage
V1 < V2, by the definition of issue order

The first two orderings imply that V2 ⇐ V1, whereas the last implies that V1 ⇐ 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 ⇐ U1
V3 reading 3 implies U1 ⇐ 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 ⇐ V1
W2 reading 2 implies V1 ⇐ 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 ⇐ V1 and V2 ⇐ 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 ⇐ 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 ⇐ V1 ⇐ V3 ⇐ U1. There is no conflicting implication that U1 ⇐ 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	
[U3] Pi:W(y,2)	[V2] Pj:R(x,1)

There are no conflicts in this sequence. V2 ⇐ U1 ⇐ U3 ⇐ V1. There is no conflicting implication that V1 ⇐ 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	[V2] Pj:MB
[U3] Pi:W(y,2)	[V3] Pj:R(x,1)

V1 reading 2 implies U3 $\Leftarrow$ V1
V3 reading 1 implies V3 $\Leftarrow$ U1
But, by transitivity, U1 $\Leftarrow$ U3 $\Leftarrow$ V1 $\Leftarrow$ 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 $\Leftarrow$ V1
V3 reading 1 implies V3 $\Leftarrow$ U1
But, by transitivity, U1 $\Leftarrow$ U3 $\Leftarrow$ V1 $\Leftarrow$ 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 $\Leftarrow$ V3
V2 $\Leftarrow$ V3 and V2 reading 3 implies V2 $\Leftarrow$ U1
V1 $\Leftarrow$ V2 and V2 $\Leftarrow$ U1 implies V1 $\Leftarrow$ U1

U3 reading 3 implies V1 $\Leftarrow$ U3
U2 $\Leftarrow$ U3 and U2 reading 2 implies U2 $\Leftarrow$ V1
U1 $\Leftarrow$ U2 and U2 $\Leftarrow$ V1 implies U1 $\Leftarrow$ V1

Both V1 ⇐ U1 and U1 ⇐ 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.

5.6.3 Implied Barriers

In Alpha, 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 or IMB instructions.)

Alpha transitions such as the following have no built-in implied memory barriers:

- Entry to PALcode
- Sending and receiving interrupts
- Returning from exceptions, interrupts, or machine checks
- Swapping context
- Invalidating the Translation Buffer (TB)

Depending on implementation choices for maintaining cache coherency, some PAL /cache implementations may have an implied IMB in the I-stream TB fill routine, but this is transparent to the non-PAL programmer.

5.6.4 Implications for Software

Software must explicitly include MB or IMB instructions in 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, an 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)

The only way to communicate shared data reliably is to write the shared data on one processor, then do an MB on that processor, 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), then do an MB, then read or update the shared data.

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

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 CPU has prepared some data buffer to be read from memory by a DMA I/O device (such as writing a buffer to disk), it must do an MB before starting the I/O, and the I/O device after receiving the start signal must logically do an MB before reading the data buffer.

This also 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 do an MB before posting a completion interrupt, and the interrupt handler software must do an MB before the data is guaranteed to be visible to the interrupted processor. Other processors must also do 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, 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 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. \This can perhaps be done in TB-miss PALcode.\

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, then do an IMB (MB if the writing processor is not going to execute the new I-stream) on that processor, 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 do an IMB, then fetch the shared I-stream.

Leaving out the first IMB(MB) removes the assurance that the shared I-stream is written before the flag is.

Leaving out the second 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 do an IMB(MB) before posting a completion interrupt, and the interrupt handler software must do an IMB before the I-stream is guaranteed to be visible to the interrupted processor. Other processors must also do IMBs before they are guaranteed to see the new I-stream.

An important special case occurs when a write is done (perhaps by an I/O device) to some physical page frame, then an IMB(MB), 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 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 [1]		
Pass ownership of process context data structure memory.	⇒	Pick up ownership of process context data structure memory. MB [2] Restore state of new process context data structure memory. Make I-stream coherent [3]. Make TB coherent [4]. : Execute code for new process that accesses memory that is not common to all processes.

MB [1] ensures that the writes done to save the state of the current process happen before the ownership is passed.

MB [2] ensures that the reads done to load the state of the new process happen after the ownership is picked up and hence are reliably the values written by the processor saving the old state. Leaving this MB out makes the code fail if an old value of the context remains in the second processor's cache and invalidates from the writes done on the first processor are not delivered soon enough.

The TB on the second processor must be made coherent with any write to the page tables that may have occurred on the first processor just before the save of the process state. This must be done with a series of TB invalidate instructions to remove any nonglobal page mapping for this process, or by assigning an ASN that is unused on the second processor to the process. One of these actions must occur sometime before starting execution of the code for the new process that accesses memory (instruction or data) that is not common to all processes. A common method is to assign a new ASN after gaining ownership of the new process and before loading its context, which includes its ASN.

The D-cache on the second processor must be made coherent with any write to the D-stream that may have occurred on the first processor just before the save of process state. This is ensured by MB [2] and does not require any additional instructions.

The I-cache on the second processor must be made coherent with any write to the I-stream that may have occurred on the first processor just before the save of process state. This can be done with an IMB PAL call 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.
		:
		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		
Send int.	⇒	Receive int.
		MB
		Access data
		:

Leaving out the MB at the beginning of the interrupt-receipt routine 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.

5.6.5 Implications for Hardware

The coherency point for physical address x is the place in the memory subsystem at which accesses to x are ordered. It may be at a main memory board, or at a cache containing x exclusively, or at the point of winning a common bus arbitration.

The coherency point for x may move with time, as exclusive access to x migrates between main memory and various caches.

MB and 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 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 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 undefined 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 undefined result R. If an instruction in the trap shadow of T uses R to calculate a subsequent register value, that register value is undefined, 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 undefined result, the stored value is undefined.
- If an instruction in the trap shadow of T uses R or any subsequent undefined result as the basis of a conditional or calculated branch, the branch target is undefined.
- If an instruction in the trap shadow of T uses R or any subsequent undefined result as the basis of an address calculation, the memory address actually accessed is undefined.

Software that is intended to bound how far the PC may advance before taking a trap, or how far an undefined 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 4.7.5.

5.8 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Changed DRAINT to TRAPB
2. Converted to SDML
3. Generalized OS specific PALcode instructions
4. Generalized OS specific multiprocessor context switching

Revision 4.0, March 29, 1991

1. Added Litmus Test 9
2. Explain what an excess data transfer is
3. Correct typing error in code sequence example for modification of atomic data structure
4. Add MB instructions to second illustrative example that specifies use of MB for multiple processor context switch
5. Note that MB and IMB do not guarantee timeliness
6. Removed reference to byte when specifying granularity of data transfer widths
7. Made minor changes to correct use of capitals and remove repeated words in the Litmus Test section

Revision 3.0, Mar 2, 1990

1. Complete rewrite of data sharing
2. Complete rewrite of read/write ordering

Revision 2.0, October 4, 1989

1. Total rewrite
2. Memory, buffer, I/O spaces removed; Physical memory regions added
3. SWP, FREEZE, and THAW removed; LDQ/L and STQ/C added
4. FAS removed; MB and NUDGE added
5. DRAIN and WAIT removed; DRAINT and /Semi-precise added

Revision 1.0, May 23, 1989

1. First Review Distribution

\

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 is that microcode will not be necessary for practical implementation. 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 resorting to a microcoded machine.

NOTE

\The hardware development groups provide and maintain the standard PALcode for a given implementation. The PALcode may be in ROM or loaded into RAM from some sort of a console load device. Many of the same trade-offs exist for PALcode that exist for microcode around patching, loading, and booting. Also, operating systems are free to provide their own PALcode rather than use the version provided by the hardware group.\

6.2 PALcode Instructions and Functions

PALcode is used to implement the following functions:

- Instructions that require complex sequencing as an atomic operation
- Instructions that require VAX-style interlocked memory access
- Privileged instructions

- Memory management control (including translation buffer (TB) management)
- Context swapping
- Interrupt and exception dispatching
- Power-up initialization and booting
- Console functions
- Emulation of instructions with no hardware support.

The Alpha architecture lets these functions be implemented in standard machine code that is resident in main memory. PALcode is written in standard machine code with some implementation-specific extensions to provide access to low-level hardware. This lets an Alpha implementation make various design trade-offs based on the hardware technology being used to implement the machine. The PALcode can abstract these differences and make them invisible to system software.

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

Part II, Operating Systems describes the Digital-supplied Alpha Privileged Architecture Library (PALcode) routines and environment. Other systems may use the Digital-supplied PALcode library or architect and implement a different library of routines. Alpha systems are required to support the replacement of Digital-defined PALcode with an operating system-specific version.

NOTE

\ The register conventions used are based on the Alpha calling standard Version 1.0. The PALcode library will track the Alpha calling standard changes as long as that is practical. \

6.3 PALcode Environment

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

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

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

6.4 Special Functions Required for PALcode

PALcode uses the Alpha instruction set for most of its operations. A small number of additional functions are needed to implement the PALcode. There are five opcodes reserved to implement PALcode functions: PALRES0, PALRES1, PALRES2, PALRES3 and PALRES4. These instructions produce an Illegal Instruction Trap if executed outside the PALcode environment.

- PALcode needs a mechanism to save the current state of the machine and dispatch into PALcode.
- PALcode needs a set of instructions to access hardware control registers.
- PALcode needs a hardware mechanism to transition the machine from the PALcode environment to the non-PALcode environment. This mechanism loads the PC, enables interrupts, enables mapping, and disables PALcode privileges.

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

- An Alpha implementation may include a read/write virtual function that allows PALcode to perform mapped memory accesses using the mapping hardware rather than providing the virtual-to-physical translation in PALcode routines. PALcode may provide a special function to do physical reads and writes and have the Alpha loads and stores continue to operate on virtual address in the PALcode environment.
- An Alpha implementation may include hardware assists for various functions—for example, saving the virtual address of a reference on a memory management error rather than having to generate it by simulating the effective address calculation in PALcode.
- An Alpha implementation may include private registers so it can function without having to save and restore the native general registers.

6.5 PALcode Effects on System Code

PALcode will have one effect on system code. Because PALcode may be resident in main memory and maintain privileged data structures in main memory, the operating system code that allocates physical memory cannot use all of physical memory.

The amount of memory PALcode requires is small, so the loss to the system is negligible.

6.6 PALcode Replacement

Alpha systems are required to support the replacement of Digital-supplied PALcode 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. The page tables documented in *Part II, Operating Systems* 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 documented in *Part II, Operating Systems* 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 must be written in a modular manner that facilitates easy replacement of major subsections. The subsections that need to be simple to replace are:

- Translation Buffer fill
- Process structure and context switch
- Interrupt and exception frame format and routine dispatch
- Privileged PALcode instructions

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. The operation of these PALcode instructions for Digital-supplied operating system implementations is described in *Part II, Operating Systems*.

Table 6–1: PALcode Instructions that Require Recognition

Mnemonic	Name
BPT	Breakpoint trap
BUGCHK	Bugcheck trap
GENTRAP	Generate trap
RDUNIQUE	Read unique value
WRUNIQUE	Write unique value

The PALcode instructions listed in Table 6–2 and described in the following sections must be supported by all Alpha implementations:

Table 6–2: Required PALcode Instructions

Mnemonic	Type	Operation
DRAINA	Privileged	Drain aborts
HALT	Privileged	Halt processor
IMB	Unprivileged	I-stream memory barrier

6.7.1 Drain Aborts

Format:

```
CALL_PAL    DRAINA                          !PALcode format
```

Operation:

```
IF  PS<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 non-existent-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. For operate instructions, this will usually mean stalling until the result register has been written. For branch instructions, this will usually mean stalling until the result register and PC have been written. For load instructions, this will usually mean stalling until the result register has been written. For store instructions, this will usually mean stalling until at least the first level in a potentially multi-level 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 multi-level 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
    halt:                {halt}
    restart/halt:        {restart/halt}
    restart/boot/halt:   {restart/boot/halt}
    boot/halt:           {boot/halt}
ENDCASE
```

Exceptions:

Privileged Instruction

Instruction mnemonics:

CALL_PAL HALT Halt Processor

Description:

The HALT instruction stops normal instruction processing, and depending on the HALT action setting, the processor may either enter console mode or the restart sequence. See *Platform Section, Chapter 4*.

NOTE

\The halt actions will be changed to match the boot and console chapters when they are done. \

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.

The cache coherency and sharing rules are described in Chapter 5.

6.8 Revision History

Revision 5.0 May 12, 1992

1. Added list of recognition-required PALcode instructions
2. Added DRAINA to list of required PALcode instructions
3. Changed privileges enabled to complete control of the machine state
4. PALcode override for TB fill routines
5. Added HALT and IMB PALcode instructions

Revision 4.1 May 12, 1992

1. Created the chapter from Sections 1.1 through 1.6 of the V4.n SRM

Chapter 7
Console Subsystem Overview (I)

On an Alpha 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 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.

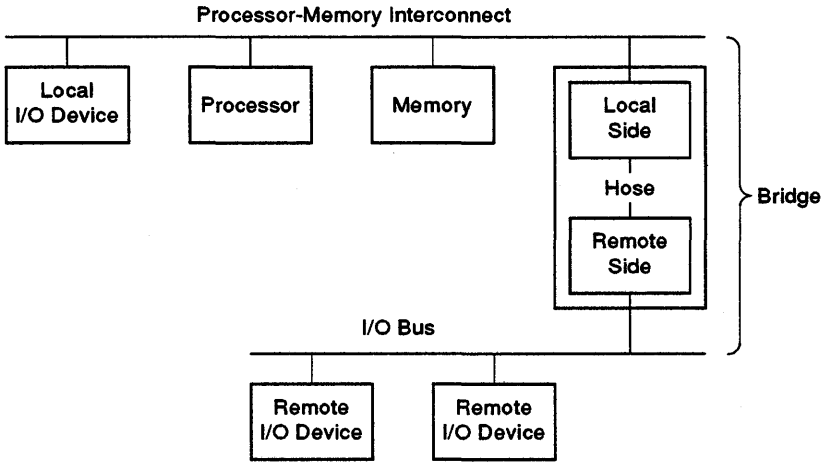
Chapter 8
Input/Output (I)

8.1 Introduction

Conceptually, Alpha systems consist of processors, memory, processor-memory interconnect (PMI), I/O buses, bridges, and I/O devices.

Figure 8–1 shows the Alpha system overview.

Figure 8–1: Alpha System Overview



As shown in Figure 8–1, processors and memory are connected by the PMI.

A bridge connects a tightly coupled I/O bus to the system, either directly to the PMI or through another tightly coupled I/O bus. A tightly coupled I/O bus is one whose address space is accessible to the processor either directly or through an I/O mailbox.

A bridge has at least a local side and a remote side, connected by a hose. The local side is electrically closer to the PMI; the remote side is electrically further.

I/O devices can be connected to the PMI or to an I/O bus. A local device connects to the PMI; a remote device connects to an I/O bus.

The following sections discuss Alpha I/O operations:

- Accesses to local I/O space are discussed in Section 8.2.
- Accesses to remote I/O space are discussed in Section 8.3.

- Reads and writes to processor memory-like regions initiated by I/O devices, or "DMAs", are discussed in Section 8.4.
- Processor interrupts requested by devices are discussed in Section 8.5.
- Bus-specific I/O accesses are discussed in Section 8.6.
- \ Some implementation-specific considerations are discussed in Section 8.7.
- Targettable interrupts are discussed in Section 8.8. \

8.2 Local I/O Space Access

Local I/O space locations may appear in either memory or non-memory-like regions. Local I/O space locations which appear in memory regions may be cached subject to the platform cache coherency scheme. See Chapter 5.

An Alpha platform need only support atomic quadword accesses. The Alpha instruction architecture requires only quadword accesses. Processor implementations may further restrict the access granularity of local I/O space. For example, a given implementation could permit addressing of only cache blocks. To support byte or word accesses to a local device, the device must be mapped into a non-memory-like region with a sparse address space. The necessary mapping is dependent on the implementation of the processor, cache, and PMI protocol. For example, the four individual bytes of a longword device control register could be mapped into the low order byte of each of four contiguous quadwords.

8.2.1 Read/Write Ordering

Access to local I/O space does not cause any implicit read/write ordering; explicit barrier instructions must be used to ensure any desired ordering. Barrier instructions must be used:

- After updating a memory-resident data structure and before writing a local I/O space location to notify the device of the updates.
- Between multiple consecutive direct accesses to local I/O space, e.g. device control registers, if those accesses are expected to be ordered at the device.

Again, note that implementations may cache not only memory-resident data structures, but also local I/O space locations.

8.3 Remote I/O Space Access

Remote I/O space locations are accessed indirectly through a memory-resident "mailbox" data structure. To post an access, the physical address of the mailbox is written into a MailBox Pointer Register (MBPR) on a local bridge side. For remote I/O space writes, the command and data are posted in the mailbox, and status is returned. For remote I/O space reads, the command is posted in the mailbox, and status and data are returned.

An Alpha system may have any number of local bridge sides. Each local side may provide connections for up to 256 hoses. Each hose may connect to a single remote

side or may connect to multiple remote sides. A single remote side may connect to one or more hoses. A bridge need not include a hose; the local and remote sides may be implemented as a single entity. A local side or an entire bridge may be incorporated into a processor board.

8.3.1 Mailbox Posting

A remote I/O space access is defined by the contents of the mailbox structure. A remote I/O space access is invoked by writing the base physical address of the mailbox structure into the appropriate bridge MailBox Pointer Register (MBPR). Each I/O bus may be associated with one and only one MBPR. A single MBPR may be associated with one or more remote I/O buses and a single bridge may have multiple MBPR registers. The MBPR appears in local I/O space.

The MBPR is accessed only with the STQ_C instruction. Flow control is achieved by the associated (per-processor) lock_flag as follows:

```
post_mbx:
        <derive PA of mailbox and load R1>
        <derive VA of MBPR and load R0>
        STQ_C   R1,R0
        BEQ     R1,wait_post_mbx
        .
        .
        .
wait_post_mbx:
        <backoff delay>
        BR      post_mbx
```

If the STQ_C lock_flag is set, the mailbox has been posted to the bridge. If the STQ_C lock_flag is clear, all MBPR resources are occupied; the MBPR write must be retried. In multi-processor configurations, this use of the STQ_C instruction affects only the local per-processor lock_flag. The state of the per-processor lock_flag of other processors is unchanged.

HARDWARE/SOFTWARE IMPLEMENTATION NOTE

The use above of the STQ_C instruction is specific to the first Alpha implementations. \(EV-3 and EV-4)\ Future implementations may use a different access mechanism. \See Section 8.7.2.\

A given remote I/O space location is uniformly accessible to all processors in a multi-processor configuration. A given hose, hence a given remote I/O bus, may be accessed via an MBPR at the same physical address from any processor. A software thread need have no knowledge of the specific processor on which it is executing.

A FIFO structure may be implemented behind each MBPR register to permit the posting of multiple outstanding mailbox operations. A set of processor-specific request queues may be implemented behind each MBPR register to ensure fair access to all processors. Any such FIFO or queue is invisible to software.

Bridge implementations must protect against lockout and ensure fair MBPR access to all processors in a multi-processor configuration. Multiple writes to an MBPR by a single processor must not be able to cause the starvation or timeout of competing writes to the same MBPR by other processors.

Multiple software threads executing at different IPLs on a single processor may cause starvation or timeout of the lower IPL threads. IPL levels are inherently unfair. \See Section 8.7.3.\

Bridge implementations must guarantee forward progress on mailbox operations regardless of direct memory access or interrupt load.

8.3.2 Mailbox Pointer Register (MBPR)

The MBPR format is shown in Figure 8–2 and described in Table 8–1.

Figure 8–2: Mailbox Pointer Register Format

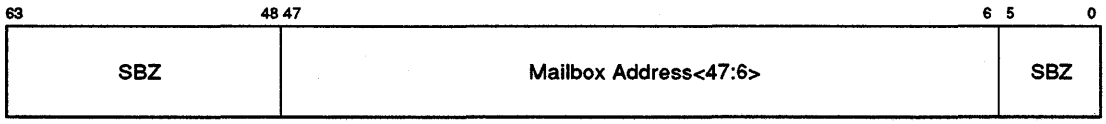


Table 8–1: Mailbox Pointer Register Format

Bit(s)	Description
<5:0>	SBZ
<47:6>	Physical address of the mailbox structure. The mailbox structure must be at least 64-byte aligned.
<63:48>	SBZ

8.3.3 Mailbox Structure

The mailbox is a 64-byte, naturally aligned, data structure. The format is shown in Figure 8–3 and described in Table 8–2.

Figure 8–3: Mailbox Data Structure Format

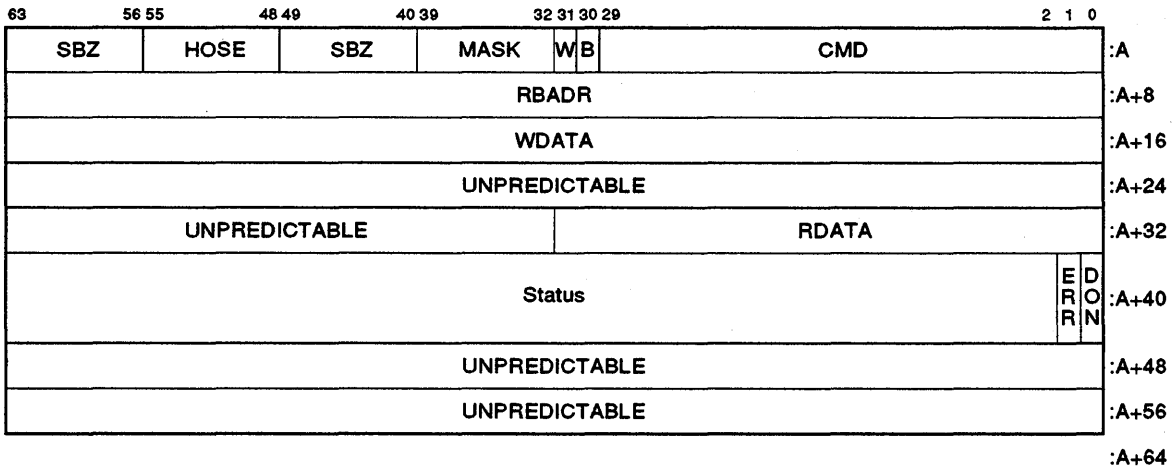


Table 8–2: Mailbox Data Structure Format

Offset	Bit(s)	Name	Description
0	<29:0>	CMD	Remote bus command. Controls the actual remote bus operation and can include fields such as address only, address width, and data width. See Section 8.6.2.
	<30>	B	Remote bridge access. If set, the command is a special or diagnostic command directed to the remote side. See Section 8.6.3.
	<31>	W	Write access. If set, the remote bus operation is a write.
	<39:32>	MASK	Disable Byte Mask. Disables bytes within the remote bus address. Mask bit <i> set causes the byte to be disabled; e.g. data byte <i> will NOT be written to the remote address. See Section 8.6.2.
	<47:40>	SBZ	
	<55:48>	HOSE	Hose. Specifies the remote bus to be accessed. Bridges may directly connect to up to 256 remote buses per hose.
	<63:56>	SBZ	
8	<63:0>	RBADR	Remote Bus Address. Contains the target address of the device on the remote bus. See Section 8.6.2.

Table 8–2 (Cont.): Mailbox Data Structure Format

Offset	Bit(s)	Name	Description
16	<63:0>	WDATA	Write Data. For write commands, contains the data to be written. For read commands, the field is not used by the bridge.
24	<63:0>		UNPREDICTABLE.
32	<31:0>	RDATA	Read Data. For read commands, contains the data returned. For write data commands, the field is UNPREDICTABLE.
	<63:32>		UNPREDICTABLE.
40	<0>	DON	Done. Indicates that the ERR, STATUS, and RDATA fields are valid; that the mailbox structure may be safely modified by host software.
	<1>	ERR	Error. If set, indicates that an error was encountered and that the STATUS field contains additional information. Valid only when DON is set. See Sections 8.3.7 and 8.3.8.
	<63:2>	STATUS	Operation completion status. Contains information specific to the bridge implementation. Valid only when DON is set. The bridge specification must include a definition of this field. See Sections 8.3.7 and 8.3.8.
48	<63:0>		UNPREDICTABLE.
56	<63:0>		UNPREDICTABLE.

8.3.4 Mailbox Access Synchronization

The ownership of the mailbox structure is exchanged between the posting software and the servicing bridge. The first 3 quadwords must be initialized by the software prior to posting the mailbox to the bridge. Once posted, the contents of the mailbox are owned by the bridge and are UNPREDICTABLE until the DON bit is set by the bridge. If the mailbox contents are altered by software prior to the DON bit becoming set, the action of the bridge and the resulting mailbox contents are UNPREDICTABLE. Once the DON bit has been set by the bridge, the mailbox contents are again owned by the software and must not be altered by the bridge. \See Section 8.7.4.\

Software use of the DON bit for synchronization is encouraged. If the DON bit is set in the mailbox at the time that the mailbox is posted, it is not possible to determine when the mailbox structure may be safely altered nor is it possible to determine when any returned information (RDATA or STATUS or ERR) becomes valid. Use of a static, not dynamically altered, mailbox structure is recommended only for true write-and-run of static data such as setting a "go" bit in a device control register.

Note that the DON bit set does NOT guarantee that a remote I/O space write has actually completed at the device. The DON bit may be set by any intervening bridge. See Section 8.3.8.

The servicing bridge ignores the contents of the DON, ERR, and STATUS fields; these fields are treated as write only.

8.3.5 Mailbox Read/Write Ordering

Mailbox accesses to a given remote bus are ordered by the MBPR and bus bridge. After posting in the MBPR, the ordering must be retained by the bridge. The bridge may reorder operations only across different hoses. Mailboxes targeted to different buses connected to the same local bridge side may occur in a sequence different from the posting order.

Mailbox operations are implicitly ordered when one and only one MBPR is used to access a given remote I/O bus. In general, there is only one path to a given remote I/O bus via a unique hose and remote side. In such configurations, the hardware must retain the ordering of mailbox accesses. In configurations in which there are multiple paths, software should order mailbox operations by using one and only one MBPR to access a given remote bus.

8.3.6 Remote I/O Space Access Granularity

The granularity of remote I/O space accesses is not symmetric:

- Mailbox reads are defined to bytes, words, and longwords.
- Mailbox writes are defined to bytes, words, longwords and quadwords.

Mailbox writes were optimized to permit efficient and atomic writes of a full 48-bit Alpha physical address.

Not all bus bridges will support all possible remote I/O space access granularities. The supported granularity will be determined by the capabilities of the remote bus and the remote bus side.

The MASK and RBADR fields are determined by the addressing and masking modes of the remote I/O bus. Invalid MASK fields, or invalid combinations of MASK and RBADR fields, will not cause ERR to be set. Error checking (if any) is done on the remote (I/O bus) side of the bridge; the local (PMI) side of the bridge employs disconnected writes. If error checking is done by the remote side of the bridge, the error is reported by an error interrupt.

On mailbox write accesses, bridges (and chains of bridges) deliver the valid WDATA, RBADR, and MASK information to the remote I/O device. The valid data may be encapsulated, along with invalid data, into larger data packets; the invalid data may simply be invalid fields from the WDATA quadword. For some remote I/O buses, the RBADR and MASK fields may be truncated or otherwise mapped.

On mailbox read accesses, bridges (and chains of bridges) deliver the valid RBADR, MASK, and command information to the remote I/O device. The bridge has no knowledge of the intended size of the read data - this is known only to the requesting software and the device, which are assumed to agree. The valid data may be encapsulated, along with invalid data, into larger data packets. Again, for some remote I/O buses, the RBADR and MASK fields may be truncated or otherwise mapped.

8.3.7 Remote I/O Space Read Accesses

The bridge must return status and data for remote I/O space reads. When the mailbox DON bit is set by the bridge, the operation has completed, and the ERR and STATUS fields may be examined. If the ERR bit is not set, the requested remote bus operation was successful and valid data was returned. If the ERR bit is set, an error was encountered and the STATUS field contains information as to the nature of the error.

Errors encountered on remote I/O space read accesses may also be reported by bridge error interrupts. The bridge side which encounters the error requests the interrupt. Thus, a non-existent hose error may be reported by the local (PMI) side of the bridge, while a non-existent remote bus address error is reported by the remote (I/O bus) side of the bridge.

Remote I/O space read accesses may be performed as follows:

```
remote_read:
        <load Rm with VA of mailbox>
        <ensure mailbox no longer in use by bridge>
        <derive and load mailbox CMD, MASK, HOSE, and RBADR fields>
        STQ     R31, 40(Rm)      ; Clear DON/ERR/STATUS fields
        MB
post_mbx:
        <derive PA of mailbox and load R1>
        <derive VA of MBPR and load R0>
        STQ_C   R1,R0
        BEQ     R1,wait_post_mbx
wait_mbxdone:
        LDQ     R0, 40(Rm)       ; Fetch STATUS/DON
        BLBS    R0, check_err    ; Branch on DON set
        <backoff delay>
        BR      wait_mbxdone
check_err:
        SRL     R0, #1, R0
        BLBS    R0, read_err
        MB
        LDQ     R0, 32(Rm)       ; Fetch RDATA
        .
        .
        .
read_err:
        <handle error>
wait_post_mbx:
        <backoff delay>
        BR      post_mbx
```

Notes:

1. The mailbox is no longer in use by a bridge whenever the DON bit has been set by the servicing bridge or is newly allocated.
2. The first barrier is required to ensure that the bridge will read the mailbox contents as updated by the processor. Any pending processor writes to the mailbox will have completed by the time that the load of the MBPR has completed.
3. The second barrier is required to ensure that the processor will read the mailbox contents as updated by the bridge. The returned data is accessed only after the DON bit is observed to be set by the servicing bridge.
4. Software need not wait for the DON bit to become set.
5. The mailbox RDATA is valid only when DON is set and ERR is clear.

8.3.8 Remote I/O Space Write Accesses

The bridge need not return status for remote I/O space writes. When the mailbox DON bit is set by the bridge, the bridge has completed access to the mailbox structure. The ERR bit and STATUS fields are testable. The actual write operation need NOT have completed at the device and the ERR bit and STATUS fields can indicate success (be cleared) even though success is not ensured. However, the ERR bit and STATUS fields, if set, do accurately report an error condition.

The actual completion of a remote I/O space write access can only be observed indirectly. Either the appropriate device state must be read back, or the device must update a memory-resident data structure and/or request an interrupt. Remote I/O space read access(es) may be posted anytime after posting the write access. Because mailbox operations to the same remote bus are guaranteed to be ordered, the read is guaranteed to occur after the write.

Errors encountered on remote I/O space write accesses are reported by bridge error interrupts. The bridge side which encounters the error requests the interrupt. Thus, a non-existent hose error may be reported by the local (PMI) side of the bridge, while a non-existent remote bus address error is reported by the remote (I/O bus) side of the bridge.

Remote I/O space write accesses may be performed as follows:

```
remote_write:
     <load Rm with VA of mailbox>
     <ensure mailbox no longer in use by bridge>
     <derive and load mailbox CMD, MASK, HOSE, and RBADR fields>
     STQ R31, 40(RM)          ; Clear DON/ERR/STATUS

     MB

post_mbx:
```

```
        <derive PA of mailbox and load R1>
        <derive VA of MBPR and load R0>
        STQ_C   R1,R0
        BEQ     R1,wait_post_mbx
         :
         :
wait_post_mbx:
        <backoff delay>
        BR      post_mbx
```

Notes:

1. The mailbox is no longer in use by a bridge whenever the DON bit has been set by the servicing bridge or is newly allocated.
2. The barrier is required to ensure that the bridge will read the mailbox contents as updated by the processor. Any pending processor writes to the mailbox will have completed by the time that the load of the MBPR has completed.
3. If the mailbox data is static, e.g. used to set a "go" bit in a device control register, the mailbox may be posted without regard to the state of the DON bit. Barriers are not required each time a static mailbox is posted, however a barrier is required after the mailbox contents are initialized and prior to its first use.

8.4 Direct Memory Accesss (DMA)

8.4.1 Access Granularity

A device or bridge side access to a memory-like region, or "DMA", is taken to be atomic when:

- It is not possible for a single device read DMA of a data structure which is updated by a single processor write to observe a partial update of that structure.
- It is not possible for a processor reading a data structure which is updated by a single device write DMA to observe a partial update of that structure.

A processor treats any memory-resident data structures which are shared with an I/O device as though the structures were shared with another processor. The processor must follow the guidelines given in *Common Architecture, Chapter 5*. Specifically, barrier instructions must be used:

1. After updating a shared memory-resident data structure and before setting an associated flag indicating that the data structure is valid.
2. After observing a newly updated flag, and prior to accessing the associated shared memory-resident data structure.

The atomic DMA size guaranteed to a local device is a function of the PMI protocol. The minimum size is an aligned hexword. Locally connected devices must obey the PMI protocol and may participate in the memory cache coherency policy. See the guidelines in *Common Architecture, Chapter 5*.

The atomic DMA size guaranteed to a remote device is a function of the remote I/O bus protocol. Remote devices are guaranteed atomic access to aligned hexwords or the remote I/O bus transfer burst size, whichever is smaller. It is the responsibility of the local bridge side to ensure the atomicity of the device DMA.

Larger atomic DMA granularity permits optimization of device control protocols. When a data structure and the associated flag are contained within a single aligned hexword, the device can update both simultaneously with a single write DMA. Similarly, the device may access both the data structure and the associated flag with a single read DMA. If the flag is valid, the data structure contains valid information; an additional read DMA is not necessary to obtain the valid data.

HARDWARE/SOFTWARE IMPLEMENTATION NOTE

The hexword write DMA size was chosen as the smallest cache block size of the first Alpha implementations \(Cobra and Flamingo)\ .

8.4.2 Read/Write Ordering

DMAs may be divided into the "control" stream and the "data" stream. These streams differ in their ordering properties.

- Control stream accesses are guaranteed to be ordered. An implicit barrier occurs before and after each access. Control stream ordering must be preserved by all bridges between a given remote I/O device and processor memory.
- Data stream DMAs may be arbitrarily reordered if permitted by the protocol of that I/O bus. No implicit barriers are associated with this stream.

A device may use control stream DMAs to ensure ordering of the data stream DMAs and of interrupt requests as seen by a processor or other device sharing the same memory-resident structures. Data stream DMAs must not be reordered with respect to control stream DMAs. Interrupt requests must not be reordered with respect to control stream DMAs.

Control stream DMAs must be used:

- As the last DMA issued to update a memory-resident data structure before requesting a processor interrupt to notify the processor of the update. This DMA ensures that any previously issued data stream DMAs become visible to the processor prior to the interrupt.
- To update any pointer or other linkage between memory-resident data structures. Consider a status buffer which is located by a status ring pointer. The status buffer may be updated with either a control or data stream DMA. The ring pointer must be updated with a control stream DMA which is issued after the last DMA used to update the status buffer.

A bridge must preserve the ordering of control stream DMAs regardless of whether the accesses are reads or writes.

The division of direct memory accesses into the control stream and the data stream is the responsibility of the device. I/O bus protocols which do not permit the separation of control and data stream DMAs must preserve the ordering of all DMAs and interrupt requests; all DMAs are considered to be control stream DMAs. Similarly, hose protocols which do not permit the separation of control and data stream DMAs must preserve the ordering of all DMAs and interrupt requests.

Bridge implementations must guarantee forward progress on all DMA operations.

8.4.3 Device Address Translation

I/O devices use only physical addresses; devices must not access page tables for the purpose of address translation. Devices are independent of any virtual memory translation scheme and processor page size.

8.5 Interrupts

An interrupt request from an I/O device consists of an interrupt priority level and an interrupt vector. Device interrupt requests are defined to be priorities 20 to 23. The interrupt vector identifies the appropriate interrupt service routine; the starting address of the interrupt service routine is obtained by using the vector as an offset from the base of the System Control Block (SCB).

All bridge implementations must maintain both the temporal order and relative priority of device interrupts. A bridge must not expedite a lower priority request if a higher priority request has been received. With one exception, a bridge must not reorder two interrupt requests at the same priority level. A bridge is permitted to expedite delivery of a fatal bridge error interrupt; this interrupt must be at IPL 23 and may take precedence over any IPL 23 device interrupts.

A bridge may prefetch the interrupt vector from an I/O device to reduce the processor overhead associated with interrupt dispatch. Vector prefetch reduces the processor latency necessary to dispatch to the interrupt service routine by reducing the delay associated with the delivery of the interrupt vector to the processor.

When a bridge delivers an interrupt from an I/O device, any pending control stream DMA writes issued by the device must have become visible to the processors. Note that due to the ordering of control stream DMAs, any data stream DMAs writes prior to the last pending control stream DMA must also have become visible to the processors.

In multi-processor configurations, interrupts may be directed to a subset of the processors in the configuration. Such targetting is implementation specific. \See Section 8.8.\

8.6 I/O Bus-Specific Mailbox Usage

\Send mail to EAGLE1::ALPHA_SRM to register a new Alpha system or bridge side.\

8.6.1 Mailbox Field Checking

Bridge sides check only implemented functions. It is the responsibility of the posting software to ensure that the mailbox data structure fields are valid and that the structure is posted correctly.

1. Local sides need not check the MASK, B, CMD, RBADR, or WDATA fields.
2. Local sides which connect to a single hose need not check the HOSE field.
3. Local sides need not pass the HOSE or W fields to the remote bridge side.
4. Remote bridge sides which do not implement masking need not check the MASK field.
5. There is no consistency checking between the W and CMD fields. If the W bit is set and the CMD field indicates a read, the result is UNPREDICTABLE. Similarly, if the W bit is clear and the CMD field indicates a write, the result is UNPREDICTABLE.
6. Remote bridge sides check only implemented CMD and RBADR bits.

8.6.2 CMD Field

The CMD field consists of two subfields:

- A remote I/O bus specific subfield.

  This subfield is common to all Alpha systems and contains the controls for a given remote bus. The common subfield must be backward compatible; all systems which connect to a given I/O bus share this subfield.

- A system-specific subfield.

  This subfield is specific to each Alpha system and contains the controls for a given bridge implementation or system-specific diagnostic functions.

The size of each is specific to the remote I/O bus. The bridge specification must include the definitions of all valid commands. This partition promotes software portability. A given device driver uses the same CMD for a given type of device access, regardless of the platform. Diagnostic software can also interpret the common field without regard to the platform on which the mailbox was posted.

8.6.3 Special Commands

The special "WHO_ARE_YOU" command (W=0, B=1, CMD=0) is common to all bridge implementations. WHO_ARE_YOU is used to determine the type of remote bridge side. In response to a mailbox operation with a WHO_ARE_YOU command and RBADR of 0, the remote bridge side returns a unique remote bus side identifier. All other commands are specific to the type of remote bus and independent of the bridge implementation.

8.7 \Implementation Considerations

8.7.1 Mailbox Selection

The choice of direct or mailbox access (local or remote I/O space) should be made after consideration of the following:

- The processor overhead associated with waiting for the return data.
- The occupancy of the processor-memory interconnect during the access to an I/O location on an I/O bus.
- The performance of the device.
- The complexity of the logic required to implement.
- The software impact.

The direct access method, with or without associated address mapping registers, is subject to the following problems on Alpha systems:

1. Access Delay.

   The I/O bus and device are typically much slower than the processor-memory interconnect and the processor.

2. Access Granularity.

   The Alpha instruction set supports only aligned quadword and longword accesses. Many I/O devices and buses require accesses that span less than four bytes; full longword accesses can generate undesired side effects.

3. Address Granularity.

   Alpha processors may have caches leading to designs which perform reads and writes to naturally aligned cache blocks. The length of a cache block is usually greater than a quadword. For memory accesses, the processor need never issue the lower address bits. Additional hardware costs would be incurred to enable the processor to access arbitrarily aligned longwords.

4. Physical Address Size

   Many I/O buses now have address spaces that exceed the Alpha address space. High performance systems need multiples of such buses. It is no longer feasible to compress or fold the I/O bus address space into a portion of the processor I/O space.

The mailbox access method addresses the above problems, but has other disadvantages. Foremost are:

- Much software has been written to perform direct (mapped) access.

  Such software must be modified to use mailbox access. Mapped I/O accesses will be compiled to longword or quadword accesses, since an Alpha compiler cannot know that any particular access is to remote I/O space. Furthermore, the LDx accesses may be reordered from the data usage. As such, it is not simply possible

to formulate an exception-based mechanism to transparently trap and handle I/O space accesses. The exception handler would have to have detailed knowledge of the device accessed to be able to resolve the appropriate access granularity.

- A mailbox operation access may require more memory accesses and processor instructions than a direct access.

  The significance of this factor depends on the relative access latencies of remote I/O space and memory. If the remote I/O space access latency is significantly longer, the effective overhead of a mailbox access will be no more than a direct access. If the remote I/O space access latency is on the order of the memory access latency, the mailbox access overhead may be significant.

  For devices which require very fast or very frequent I/O space accesses, e.g. frame buffers, mailbox accesses can be expected to give unacceptable system performance. Additional hardware such as a companion DMA engine or attached local processor must be coupled to the device.

To promote portability, software should be written to accommodate a bridge. It is recommended that ALL I/O location reads and writes are made through subroutines. Parameters to these routines should include all the fields necessary to use a mailbox, see Section 8.3.3.

8.7.2 Mailbox Pointer Register Flow Control Selection

Each Mailbox Pointer (MBPR) register represents a resource to the processor. Either that resource must appear to be infinite, or a flow control mechanism is necessary.

The MBPR resources appear to be infinite when, barring hardware errors, posting a mailbox access is guaranteed to succeed. A sufficiently deep FIFO structure implemented behind the MBPR register could appear infinite. The depth of the FIFO will be a function of the number of I/O devices to be supported and the access characteristics of those devices. A hardware mechanism for backoff-retry access to the MBPR incorporated in the PMI protocol could also provide such a guarantee.

A flow control mechanism for MBPR register accesses must be atomic. The MBPR is accessed by code threads which execute at multiple IPLs. A single software MBPR ownership flag would lead to priority inversion and/or deadlock. A higher IPL code thread executing on one processor will block if the flag is owned by a lower IPL code thread executing on a different processor.

The MBPR register access flow control mechanism should not add significant overhead to critical code paths. Performing MBPR accesses only at IPL 31 or via dedicated PALcode can have significant system performance implications. Statically allocating some number of MBPR resources (FIFO entries) per IPL and/or per processor requires that the software thread determine the IPL/processor execution environment. Note that such static allocation schemes are not guaranteed to be portable between Alpha systems.

The first Alpha implementations use a single STQ_C instruction and the associated lock_flag to implement MBPR register access flow control. This is an implementation choice and not architected. Subsequent implementations may select

other mechanisms, particularly since this use of STQ_C may have performance implications.

IMPLEMENTATION NOTE

As an example, consider a processor with virtual caches. Virtual address translation would be required on all STQ_C instructions to differentiate the MBPR accesses from the memory accesses; the translation overhead would slow all STQ_C instructions.

8.7.3 Mailbox Starvation

The MBPR register represents a shared system resource. Software which issues mailbox accesses should use that resource in a manner which guards against starvation or access lockout.

Consider two software threads each issuing repeated mailbox accesses. There are three cases of interest:

1. Each thread is executing on a unique processor in a multi-processor configuration. The bridge hardware implementation will provide fair MBPR access to each thread. Neither thread can cause the starvation of the other.
2. Both threads are scheduled for execution at non-elevated IPL (IPL 0) on the same processor in a multi-processor configuration or on the only processor in a uni-processor configuration. The operating system software scheduling policy may provide fair MBPR access to each thread, or may allow either thread to cause the starvation of the other.
3. Both threads are scheduled for execution on the same processor in a multi-processor configuration or on the only processor in a uni-processor configuration and at least one of the threads is scheduled for execution at elevated IPL (IPL > 0). The thread which executes at the highest IPL can cause starvation of the thread executing at the lower IPL level. If both threads are scheduled to execute at the same IPL, either thread can cause starvation of the other.

Software threads which execute at high IPL for extended periods can have severe system performance implications. Remote I/O space accesses are inherently slow with respect to processor speeds; remote I/O accesses can easily take in excess of 1000 instructions. Software which spins at high IPL waiting for the DON bit or repeatedly posting mailbox accesses may execute for extended periods and cause blockage of other event delivery.

8.7.4 Mailbox Structure Synchronization Properties

As explained in Section 8.3.4, the software and the servicing bridge may synchronize their accesses to the mailbox structure by using the DON bit.

Bus bridge implementations may overwrite the full mailbox structure when setting the DON bit. The bridge may perform a full 64-byte write to the mailbox structure rather than a single quadword write or 32-byte write. If the bridge writes into the

first hexword, the original mailbox contents must be restored; the bridge must not cause the contents of the first hexword to be altered.

Software must not alter the mailbox contents at any time after writing the MBPR and prior to observing the DON bit set. Any such changes may or may not be observed by the bridge. Any such changes may or may not be overwritten by the bridge. The resulting remote bus access and the resulting mailbox contents are UNPREDICTABLE.

Software may chose to ignore the DON bit if the contents of the mailbox structure are truly static. Software may post the same mailbox repeatedly. Bridge implementations must be able to correctly access the same mailbox in the event of back-to-back MBPR writes with the same mailbox address. Note that in this case, the contents of the DON, ERR, and STATUS fields are UNPREDICTABLE.

8.7.5 I/O Device Properties

Devices should be designed such that register accesses in the main code path can be retried with minimum knowledge of the nature of the device or the side effects of the access. Read accesses should not be used to signal a device to poll a command queue, increment a counter or pointer, or initiate an I/O operation. This permits the software error recovery from transient errors to occur outside the main execution thread of the device driver.

Device designs are strongly encouraged NOT to require reads from device registers during normal operation. Such reads can easily take in excess of 1000 instruction cycles and become a major performance impact in a very high speed system.

Device designs are strongly encouraged NOT to require multiple back-to-back writes to device registers during normal operation. Such writes can lead to congestion at the MBPR, thus causing at least the issuing processor to wait. Such congestion can become a major performance impact in a very high speed system.

The mailbox protocol does not provide any indication that a write has actually completed at the device. Device designs which use writes to registers to initiate device actions are strongly encouraged to include a mechanism in the control protocol to detect a lost signal or otherwise simply recover from a delayed notification.

8.7.6 Implications of Memory Accesses by Devices

Devices access memory for the exchange of command, status, and data with the processor. Repeated processor accesses to non-cached locations, even if the location is resident on the processor-memory interconnect, may have a negative performance impact in a very high speed system. Such accesses should be replaced with cacheable (e.g. memory) accesses wherever possible.

Bridges and local devices may incorporate physical memory buffers and participate in the cache coherency policy. A bridge implementation which includes a cache may not permit hits under misses for control stream DMA reads. Such reordering would prohibit a device from issuing two back-to-back control stream DMA reads to access a single data structure since the cache hit could contain outdated data.

The dominant component of delay in a read DMA request by a remote I/O device may be the memory access latency rather than the data transmission time. Fewer, larger, memory accesses are preferable to many small accesses. Also, write control stream DMAs to less than a full cache block may consume PMI resources if the bridge must do a read-modify-write.

The device control protocol data structures should be compact and naturally aligned. Note that this may require some memory-to-memory copies by the processor. Small memory reads which must be serialized should be minimized; a common cause of such reads is when the device chases a collection of pointers.

Device control protocols must NOT make use of memory interlocks. Devices are not guaranteed emulation of the VAX interlocked instructions such as INSQTI/REMQTI. Use of functionality equivalent to LDx_L/STx_C need not be supported by bridges and is not recommended for remote devices.

8.7.7 Interrupts

A device interrupt allows a device or bridge to signal processors for various reasons, often including the following:

- Device solicitations for new I/O operations.
- Operation completion.
- Availability of operation status.
- Error occurrences.
- Non-host-originated software-relevant changes in device or bridge state or identity.

Device port protocols are strongly encouraged to minimize the use of interrupts, since interrupts have an expensive, and increasing, performance impact. The performance impact is due to many factors. Interrupts cause processor pipeline breaks and the execution of diverse short code threads which lower the effective cache and translation buffer hit rate. Instruction execution is slowed during the time required to obtain the hardware interrupt vector.

Interrupts in an Alpha system may target one or more processors. While multiple processors may respond, only one will actually transfer control to the interrupt service routine.

Conceptually, for a device on an I/O bus, the interrupt protocol is:

1. The device issues an interrupt request to the I/O module. The request specifies at least an interrupt level, corresponding to IPL 20 to 23.
2. The bridge may prefetch the interrupt vector. This reduces the latency associated with the delivery to the responding processor.
3. The bridge issues an interrupt request to some subset of the processors in the system. If the PMI protocol permits, the vector may be forwarded with the interrupt request. The interrupt is now outstanding.

4. When the IPL of an interrupted processor is lower than that of one or more outstanding interrupts, the processor will obtain a hardware interrupt vector if it does not already have one. The first processor to request a vector from a bridge or device will obtain the next pending vector. The "next pending" vector is determined by the IPL and time sequence order in which interrupts became pending at the bridge or device. The bridge or device does not reorder interrupts with the exception of a fatal bridge error interrupt; the latter occurs only at IPL 23.

5. The processor obtaining the hardware interrupt vector uses it as an offset from the base of the System Control Block. The System Control Block element contains the software interrupt vector, which is the starting address of the interrupt service routine. The software interrupt vector is referred to as the interrupt vector in *Part II, Operating Systems*. The processor transfers control to this address.

As a minimum, there should be no more than one interrupt on average for each operation carried out by the device.

8.8 Targettable Interrupts

In multi-processor configurations, interrupts may be directed, or targetted, to a subset of the processors in the configuration. The targetted subset may include one or more of the processors. Different interrupt sources, e.g. bridges, hoses, or devices, may be targetted to a different subset. Such targetting is implementation specific.

Implementations which target interrupts must include mechanisms for handling the precedence of the bridge or device error interrupt. When interrupts can be taken by one of many processors, an error interrupt may be taken by one processor while a success interrupt is taken by another processor. If the event which generated the error interrupt is related to the event which generated the success interrupt, the error interrupt must be fully serviced before the success interrupt can be serviced.

As an example, consider a device which issues a control stream DMA write, then requests a completion (success) interrupt. If a bridge incurs an error on that DMA, the bridge may discard the DMA data and request an error interrupt. If the two interrupts are serviced simultaneously on two different processors, the software thread servicing the success interrupt may take incorrect action based on faulty (stale) data. The error condition must be evaluated prior to permitting the success code thread to execute.

8.9 \Revision History:

Revision 5.0, May 12, 1992

1. Changed 'widget' to 'device'
2. Split chapter such that Sections 1.1 through the text part of 1.6.3 are now external Chapter 8 of the Common Section, Table 1-3 and all text/tables through 1.6.3.2 (Futurebus+...) are placed in Appendix D, and 1.7 (Implementation Considerations) to end of chapter are internal (backslash) Chapter 8 of the Common Section
3. Changed hex IPLs to decimal
4. Made specified internal references external
5. Added ECO #22
6. Converted to SDML
7. Made all 'unpredictable' to 'UNPREDICTABLE'
8. Changed SLL to SRL under 'check error:' in remote read psuedocode
9. Removed all revision history prior to Rev 4.0, 29 March 1991

Revision 4.1, August 12, 1991

1. Renumbered Chapter to #11 with inclusion of Console ECO #15

Revision 4.0, March 29, 1991

1. Inclusion in REV 4.0 of the SRM numbering to assume SRM version values

\

OpenVMS Alpha Software (II)

This section describes how the OpenVMS operating system relates to the Alpha architecture and contains the following chapters:

- Chapter 1, Introduction to OpenVMS Alpha (II)
- Chapter 2, OpenVMS PALcode Instruction Descriptions (II)
- Chapter 3, OpenVMS Memory Management (II)
- Chapter 4, OpenVMS Process Structure (II)
- Chapter 5, OpenVMS Internal Processor Registers, (II)
- Chapter 6, OpenVMS Exceptions, Interrupts, and Machine Checks (II)

Contents

Chapter 1 Introduction to OpenVMS Alpha (II)

Chapter 2 OpenVMS PALcode Instruction Descriptions (II)

Chapter 3 OpenVMS Memory Management (II)

Chapter 4 OpenVMS Process Structure (II)

Chapter 5 OpenVMS Internal Processor Registers, (II)

Chapter 6 OpenVMS Exceptions, Interrupts, and Machine Checks (II)

Figures

Tables

Chapter 1

Introduction to OpenVMS Alpha (II)

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

1.1 Register Usage

Besides those registers described in *Part I, Common Architecture*, OpenVMS defines the registers described in the following sections.

1.1.1 Processor Status

The Processor Status (PS) is a special register that contains the current status of the processor. It can be read by the CALL_PAL RD_PS instruction. The software field (PS<SW>) can be written by the CALL_PAL WR_PS_SW routine. See Chapter 6 for a description of the PS register.)

1.1.2 Stack Pointer (SP)

Integer register R30 is the Stack Pointer (SP).

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

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

1.1.3 Internal Processor Registers (IPRs)

The IPRs provide an architected mapping to internal hardware or provide other specialized uses. They are available only to privileged software through PALcode routines and allow OpenVMS to interrogate or modify system state. The IPRs are described in Chapter 5.

1.2 \Revision History

Revision 1.0, May 12, 1992

- Created for SRM Version 5
- First review distribution

\

Chapter 2

OpenVMS PALcode Instruction Descriptions (II)

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

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

Table 2–1 lists all the unprivileged and privileged OpenVMS PALcode instructions and the section in this chapter in which they are described.

Table 2–1: OpenVMS PALcode Instructions

Unprivileged OpenVMS PALcode Instructions

Mnemonic	Operation	Section
AMOVRM	Atomic move register/memory	Section 2.4
AMOVRR	Atomic move register/register	Section 2.4
BPT	Breakpoint	Section 2.1
BUGCHK	Bugcheck	Section 2.1
CHME	Change mode to executive	Section 2.1
CHMK	Change mode to kernel	Section 2.1
CHMS	Change mode to supervisor	Section 2.1
CHMU	Change mode to user	Section 2.1
GENTRAP	Generate software trap	Section 2.1
IMB	I-stream memory barrier	*Common Architecture, Chapter 6*
INSQxxx	Insert in specified queue	Section 2.3
PROBER	Probe read access	Section 2.1
PROBEW	Probe write access	Section 2.1
RD_PS	Read processor status	Section 2.1

Table 2–1 (Cont.): OpenVMS PALcode Instructions

Unprivileged OpenVMS PALcode Instructions

Mnemonic	Operation	Section
READ_UNQ	Read unique context	Section 2.5
REI	Return from exception or interrupt	Section 2.1
REMQxxx	Remove from specified queue	Section 2.3
RSCC	Read system cycle counter	Section 2.1
SWASTEN	Swap AST enable	Section 2.1
WRITE_UNQ	Write unique context	Section 2.5
WR_PS_SW	Write processor status software field	Section 2.1

Privileged OpenVMS PALcode Instructions

Mnemonic	Operation	Section
CFLUSH	Cache flush	Section 2.6
DRAINA	Drain aborts	*Common Architecture, Chapter 6*
HALT	Halt processor	*Common Architecture, Chapter 6*
LDQP	Load quadword physical	Section 2.6
MFPR	Move from processor register	Section 2.6
MTPR	Move to processor register	Section 2.6
STQP	Store quadword physical	Section 2.6
SWPCTX	Swap privileged context	Section 2.6

2.1 Unprivileged General OpenVMS PALcode Instructions

The general unprivileged instructions in this section, together with those in Sections 2.3, 2.4, and 2.5, provide support for the underlying OpenVMS Alpha model.

Table 2–2: Unprivileged General OpenVMS PALcode Instruction Summary

Mnemonic	Operation
BPT	Breakpoint
BUGCHK	Bugcheck
CHME	Change mode to executive
CHMK	Change mode to kernel
CHMS	Change mode to supervisor
CHMU	Change mode to user
GENTRAP	Generate software trap
IMB	I-stream memory barrier See *Common Architecture, Chapter 6*
PROBER	Probe read access
PROBEW	Probe write access
RD_PS	Read processor status
REI	Return from exception or interrupt
RSCC	Read system cycle counter
SWASTEN	Swap AST enable
WR_PS_SW	Write processor status software field

2.1.1 Breakpoint

Format:

```
CALL_PAL    BPT                                    !PALcode format
```

Operation:

```
{initiate BPT exception with new_mode=kernel}
```

Exceptions:

Kernel Stack Not Valid Halt

Instruction Mnemonics:

```
CALL_PAL    BPT          Breakpoint
```

Description:

The BPT instruction is provided for program debugging. It switches to Kernel mode and pushes R2..R7, the updated PC, and PS on the Kernel stack. It then dispatches to the address in the Breakpoint SCB vector. See Section 6.3.3.2.1.

2.1.2 Bugcheck

Format:

```
CALL_PAL    BUGCHK                          !PALcode format
```

Operation:

```
{initiate BUGCHK exception with new_mode=kernel}
```

Exceptions:

Kernel Stack Not Valid Halt

Instruction Mnemonics:

```
CALL_PAL    BUGCHK      Bugcheck
```

Description:

The BUGCHK instruction is provided for error reporting. It switches to Kernel mode and pushes R2..R7, the updated PC, and PS on the Kernel stack. It then dispatches to the address in the Bugcheck SCB vector. See Section 6.3.3.2.2.

2.1.3 Change Mode Executive

Format:

CALL_PAL CHME !PALcode format

Operation:

```
tmp1 ← MINU( 1, PS<CM>)
{initiate CHME exception with new_mode=tmp1}
```

Exceptions:

Kernel Stack Not Valid Halt

Instruction Mnemonics:

CALL_PAL CHME Change Mode to Executive

Description:

The CHME instruction lets a process change its mode in a controlled manner.

A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PC and PS are pushed onto the selected stack. The saved PC addresses the instruction following the CHME instruction. Registers R22, R23, R24, and R27 are available for use by PALcode as scratch registers. The contents of these registers are not preserved across a CHME.

2.1.4 Change Mode to Kernel

Format:

```
CALL_PAL    CHMK                                    !PALcode format
```

Operation:

```
{initiate CHMK exception with new_mode=kernel}
```

Exceptions:

Kernel Stack Not Valid Halt

Instruction Mnemonics:

```
CALL_PAL    CHMK        Change Mode to Kernel
```

Description:

The CHMK instruction lets a process change its mode to kernel in a controlled manner.

A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PC, and PS are pushed onto the kernel stack. The saved PC addresses the instruction following the CHMK instruction. Registers R22, R23, R24, and R27 are available for use by PALcode as scratch registers. The contents of these registers are not preserved across a CHMK.

2.1.5 Change Mode Supervisor

Format:

```
CALL_PAL    CHMS                                    !PALcode format
```

Operation:

```
tmp1 ←  MINU( 2, PS<CM>)
{initiate CHMS exception with new_mode=tmp1}
```

Exceptions:

Kernel Stack Not Valid Halt

Instruction Mnemonics:

```
CALL_PAL    CHMS        Change Mode to Supervisor
```

Description:

The CHMS instruction lets a process change its mode in a controlled manner.

A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PC, and PS are pushed onto the selected stack. The saved PC addresses the instruction following the CHMS instruction.

2.1.6 Change Mode User

Format:

```
CALL_PAL    CHMU                                    !PALcode format
```

Operation:

```
{initiate CHMU exception with new_mode=PS<CM>}
```

Exceptions:

Kernel Stack Not Valid Halt

Instruction Mnemonics:

```
CALL_PAL    CHMU        Change Mode to User
```

Description:

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

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

The CALL_PAL CHMU instruction is provided for VAX compatibility only.

2.1.7 Generate Software Trap

Format:

CALL_PAL GENTRAP !PALcode format

Operation:

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

Exceptions:

Kernel Stack Not Valid Halt

Instruction Mnemonics:

CALL_PAL GENTRAP Generate Software Trap

Description:

The GENTRAP instruction is provided for reporting runtime software conditions. It switches to Kernel mode, and pushes R2...R7, the updated PC and PS on the Kernel stack. It then dispatches to the address in the GENTRAP SCB Vector. See Section Section 6.6.

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

2.1.8 Probe Memory Access

Format:

```
CALL_PAL   PROBE                                    !PALcode format
```

Operation:

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

first ← R16
last  ← {R16+R17}

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

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

Exceptions:

Translation Not Valid

Instruction Mnemonics:

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

Description:

The PROBE instruction checks the read or write accessibility of the first and last byte specified by the base address and the signed offset; the bytes in between are not checked.

System software must check all pages between the two bytes if they are to be accessed. If both bytes are accessible, PROBE returns the value 1 in R0; otherwise, PROBE returns 0. The Fault On Read and Fault On Write PTE bits are not checked. A Translation Not Valid exception is signaled only if the the mapping structures can not be accessed. A Translation Not Valid exception is signaled only if the first or second level PTE is invalid.

The protection is checked against the less privileged of the modes specified by R18<1:0> and the Current Mode (PS<CM>). See Section 6.2 for access mode encodings.

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

2.1.9 Read Processor Status

Format:

```
CALL_PAL    RD_PS                          !PALcode format
```

Operation:

```
R0 ← PS
```

Exceptions:

None

Instruction Mnemonics:

```
CALL_PAL    RD_PS        Read Processor Status
```

Description:

The RD_PS instruction returns the Processor Status (PS) in register R0. The Processor Status is described in Section 6.2. The PS<SP_ALIGN> field is always a zero on a RD_PS.

2.1.10 Return from Exception or Interrupt

Format:

```
CALL_PAL   REI                                    !PALcode format
```

Operation:

```
! See Chapter 6
!  for information on interrupted registers

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

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

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

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

```
R2 ←  tmp1
R3 ←  tmp2
R4 ←  tmp3
R5 ←  tmp4
R6 ←  tmp5
R7 ←  tmp6
PC ←  tmp7
PS ←  tmp8 <12:00>
{Initiate interrupts or AST interrupts that are now pending}
```

Exceptions:

Access Violation

Fault on Read

Illegal Operand

Kernel Stack Not Valid Halt

Translation Not Valid

Instruction Mnemonics:

CALL_PAL	REI	Return from Exception or Interrupt

Description:

The REI instruction pops the PS, PC, and saved R2...R7 from the current stack and holds them in temporary registers.

The new PS is checked for validity and consistency. If it is invalid or inconsistent, an illegal operand exception occurs; otherwise the operation continues. A kernel to nonkernel REI with a new PS<IPL> not equal to zero may yield UNDEFINED results.

The current stack pointer is then saved and a new stack pointer is selected according to the new PS<CM> field. R2 through R7 are restored using the saved values held in the temporary registers. A check is made to determine if an AST or other interrupt is pending (see Section 6.7.6).

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

When an REI is issued, the current stack must be writable from the current mode or an Access Violation may occur.

IMPLEMENTATION NOTE

This is necessary so that an implementation can choose to clear the lock_flag by doing a STx_C to above the top-of-stack after popping PS, PC, and saved R2..R7 off the the current stack.

2.1.11 Read System Cycle Counter

Format:

```
CALL_PAL    RSCC                                    !PALcode format
```

Operation:

```
R0 ←  {System Cycle Counter}
```

Exceptions:

None

Instruction Mnemonics:

```
CALL_PAL    RSCC        Read System Cycle Counter
```

Description:

The RSCC instruction writes register R0 with the value of the system cycle counter. This counter is an unsigned 64-bit integer that increments at the same rate as the process cycle counter. The cycle counter frequency, which is the number of times the system cycle counter gets incremented per second rounded to a 64-bit integer, is given in the HWRPB. \ (See *Platform Section, Chapter 3*). \

The system cycle counter is suitable for timing a general range of intervals to within 10% error and may be used for detailed performance characterization. It is required on all implementations. SCC is required for every processor, and each processor in a multiprocessor system has its own private, independent SCC.

Notes:

1. Processor initialization starts the SCC at 0.
2. SCC is required for every processor and each processor in a multiprocessor system has its own private, independent SCC.
3. SCC is monotonically increasing. On the same processor, the values returned by two successive reads of SCC must either be equal or the value of the second must be greater (unsigned) than the first.
4. SCC ticks are never lost so long as the SCC is accessed at least once per each PCC overflow period (2**32 PCC increments) during periods when the hardware clock interrupt remains blocked. The hardware clock interrupt is blocked whenever the IPL is at or above CLOCK_IPL or whenever the processor enters console I/O mode from program I/O mode.

5. The 64-bit SCC may be constructed from the 32-bit PCC hardware counter and a 32-bit PALcode software counter. As part of the hardware clock interrupt processing, PALcode increments the software counter whenever a PCC wrap is detected. Thus, SCC ticks may be lost only when PALcode fails to detect PCC wraps. In a machine where the PCC is incremented at a 1 nsec rate, this may occur when hardware clock interrupts are blocked for greater than 4 seconds.

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

2.1.12 Swap AST Enable

Format:

```
CALL_PAL    SWASTEN                          !PALcode format
```

Operation:

```
R0 ←  ZEXT(ASTEN<PS<CM>>)
ASTEN<PS<CM>> ←  R16<0>
{check for pending ASTs}
```

Exceptions:

None

Instruction Mnemonics:

CALL_PAL SWASTEN Swap AST Enable for Current Mode

Description:

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

A check is made to determine if an AST interrupt is pending (see Section 6.7.6.6).

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

2.1.13 Write Processor Status Software Field

Format:

```
CALL_PAL    WR_PS_SW                        !PALcode format
```

Operation:

```
PS<SW> ←  R16<1:0>
```

Exceptions:

None

Instruction Mnemonics:

CALL_PAL WR_PS_SW Write Processor Status Software Field

Description:

The WR_PS_SW instruction writes the Processor Status software field (PS<SW>) with the low order two bits of R16. The Processor Status is described in Section 6.2.

2.2 OpenVMS Alpha Queue Data Types

The following sections describe the queue data types that are manipulated by the OpenVMS queue PALcode. Section 2.3 describes the PALcode instructions that perform the manipulation.

2.2.1 Absolute Longword Queues

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

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

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

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

2.2.2 Self-Relative Longword Queues

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

An empty longword queue is specified by its header at address H. Since the longword queue is empty, the self-relative links are zero, as shown in Figure 2–6.

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

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

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

Figures 2–7, 2–8, and 2–9, respectively, illustrate the results of subsequent insertion of an entry at address B at the head, insertion of an entry at address A at the tail, and insertion of an entry at address C at the tail.

Figures 2–9, 2–8, and 2–7 (in that order) illustrate the effect of removal at the tail and removal at the head.

Figure 2–1: Empty Absolute Longword Queue

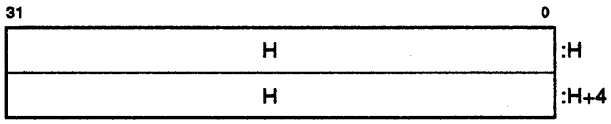


Figure 2–2: Absolute Longword Queue with One Entry

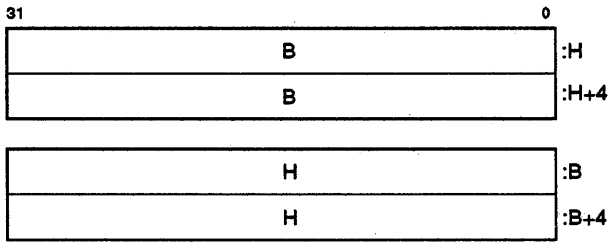


Figure 2–3: Absolute Longword Queue with Two Entries

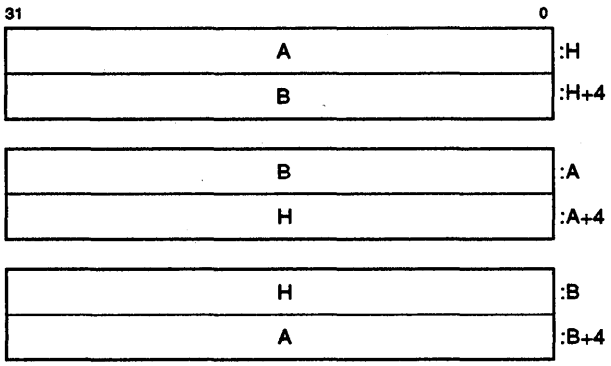


Figure 2–4: Absolute Longword Queue with Three Entries

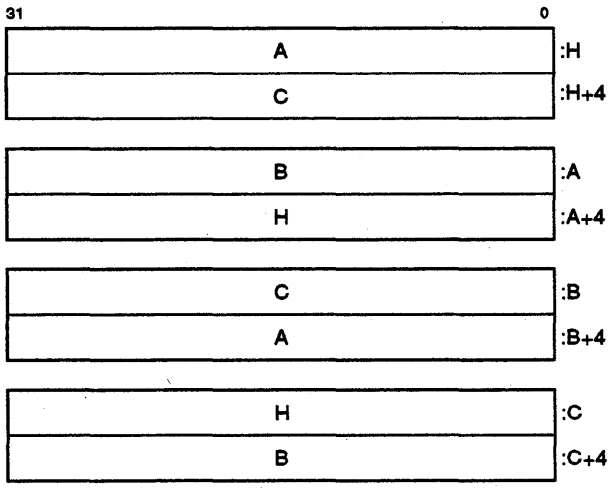


Figure 2–5: Absolute Longword Queue with Three Entries after Removing the Second Entry

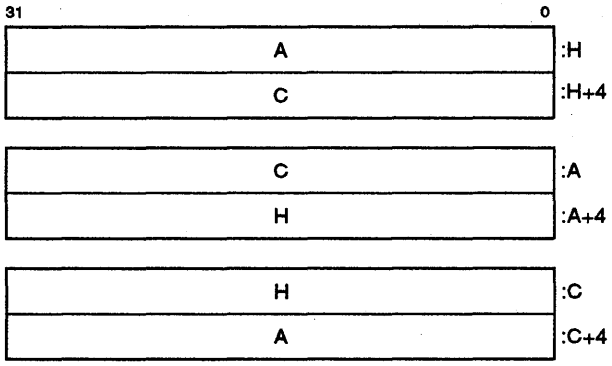


Figure 2–6: Empty Self-Relative Longword Queue

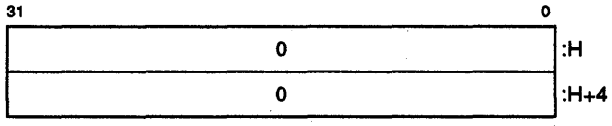


Figure 2–7: Self-Relative Longword Queue with One Entry

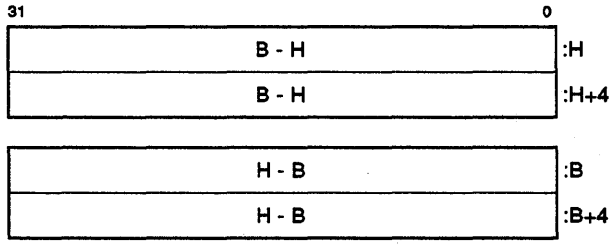


Figure 2–8: Self-Relative Longword Queue with Two Entries

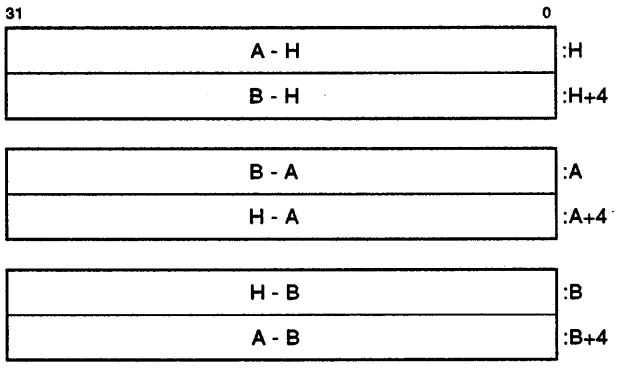
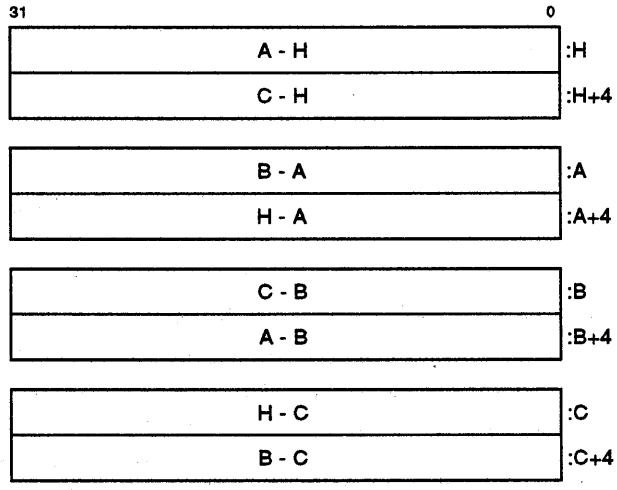


Figure 2–9: Self-Relative Longword Queue with Three Entries



2.2.3 Absolute Quadword Queues

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

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

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

An empty quadword queue is specified by its header at address H, as shown in Figure 2–10. If an entry at address B is inserted into an empty quadword queue (at either the head or tail), the quadword queue shown in Figure 2–11 results. Figures 2–12, 2–13, and 2–14, respectively, illustrate the results of subsequent insertion of an entry at address A at the head, insertion of an entry at address C at the tail, and removal of the entry at address B.

2.2.4 Self-Relative Quadword Queues

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

An empty quadword queue is specified by its header at address H. Since the quadword queue is empty, the self-relative links are zero, as shown in Figure 2–15.

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

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

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

Figures 2–16, 2–17, and 2–18, respectively, illustrate the results of subsequent insertion of an entry at address B at the head, insertion of an entry at address A at the tail, and insertion of an entry at address C at the tail.

Figures 2–18, 2–17, and 2–16, (in that order) illustrate the effect of removal at the tail and removal at the head.

Figure 2–10: Empty Absolute Quadword Queue

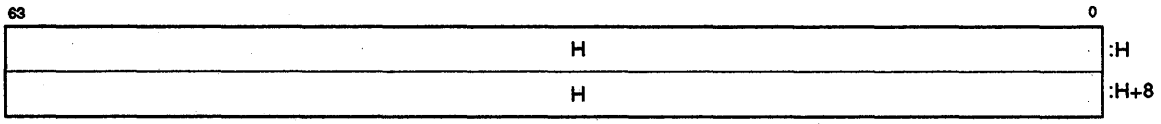


Figure 2–11: Absolute Quadword Queue with One Entry

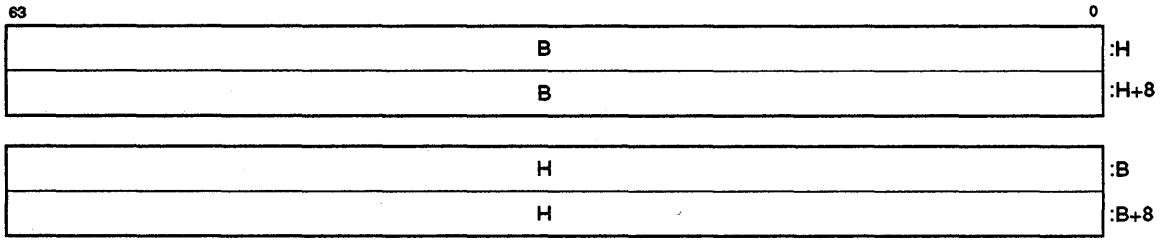


Figure 2–12: Absolute Quadword Queue with Two Entries

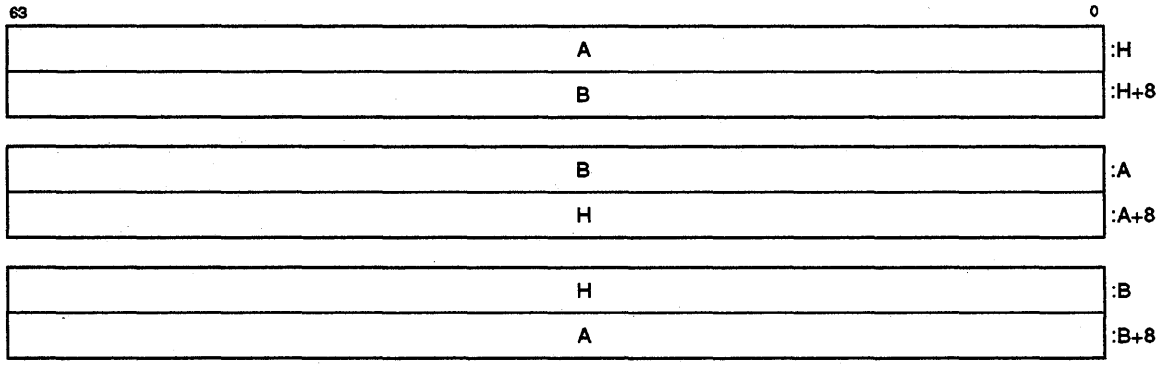


Figure 2–13: Absolute Quadword Queue with Three Entries

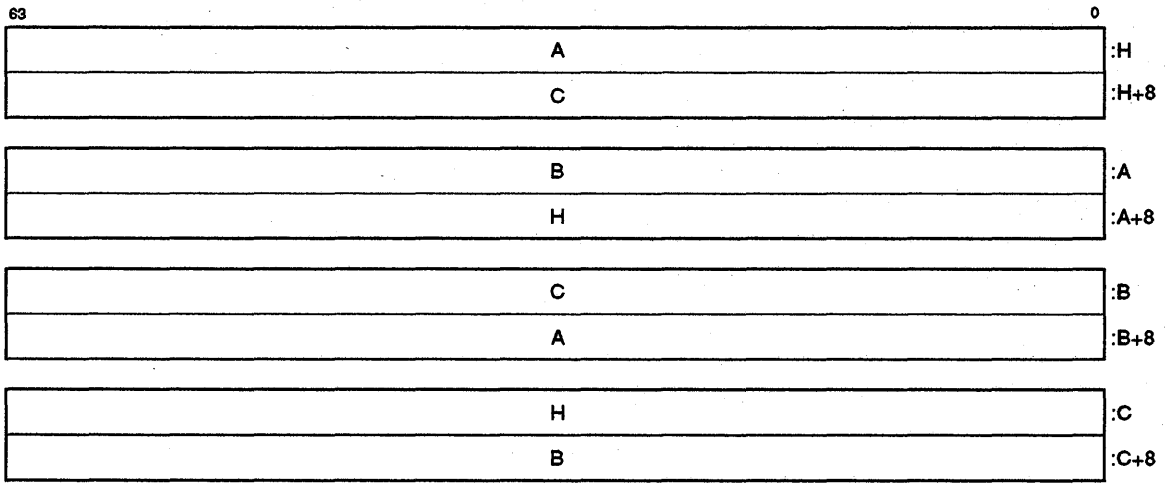


Figure 2–14: Absolute Quadword Queue with Three Entries After Removing the Second Entry

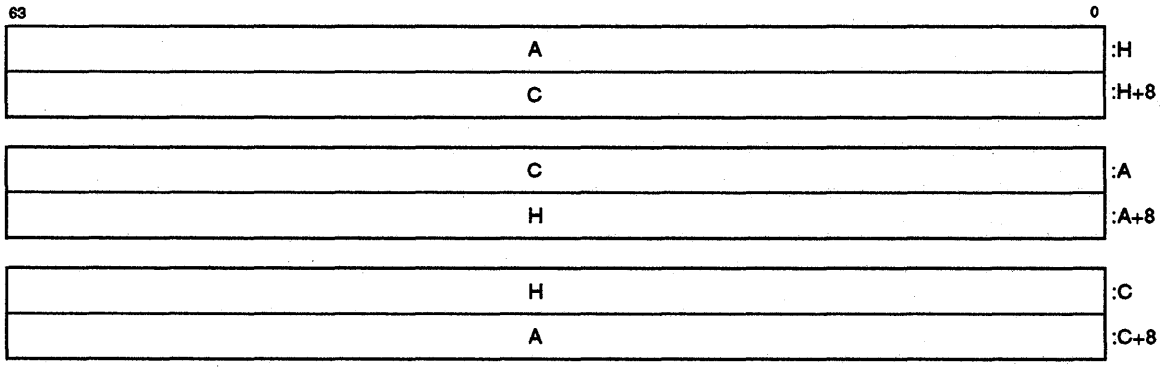


Figure 2–15: Empty Self-Relative Quadword Queue

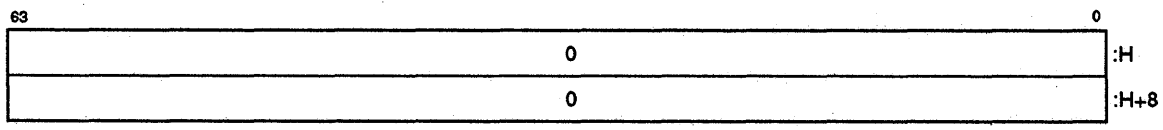


Figure 2–16: Absolute Quadword Queue with One Entry

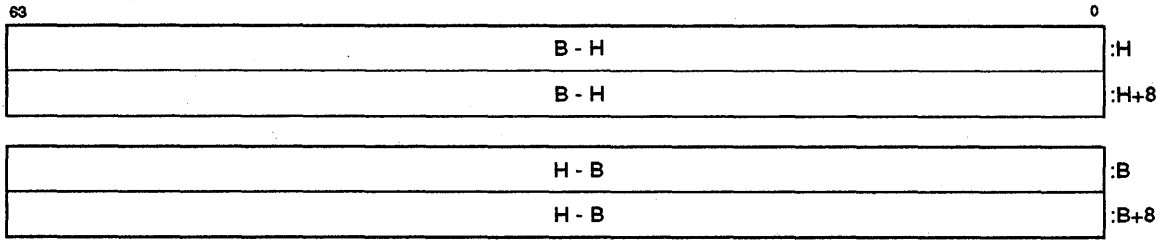


Figure 2–17: Self-Relative Quadword Queue with Two Entries

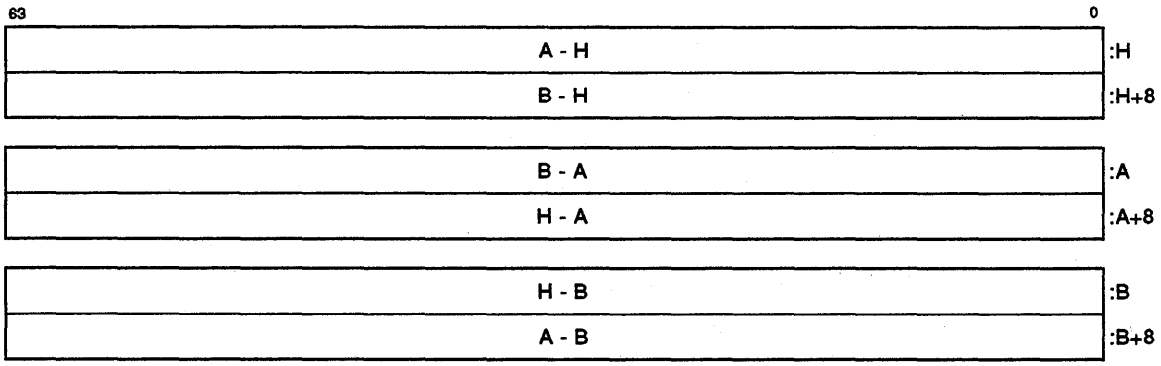
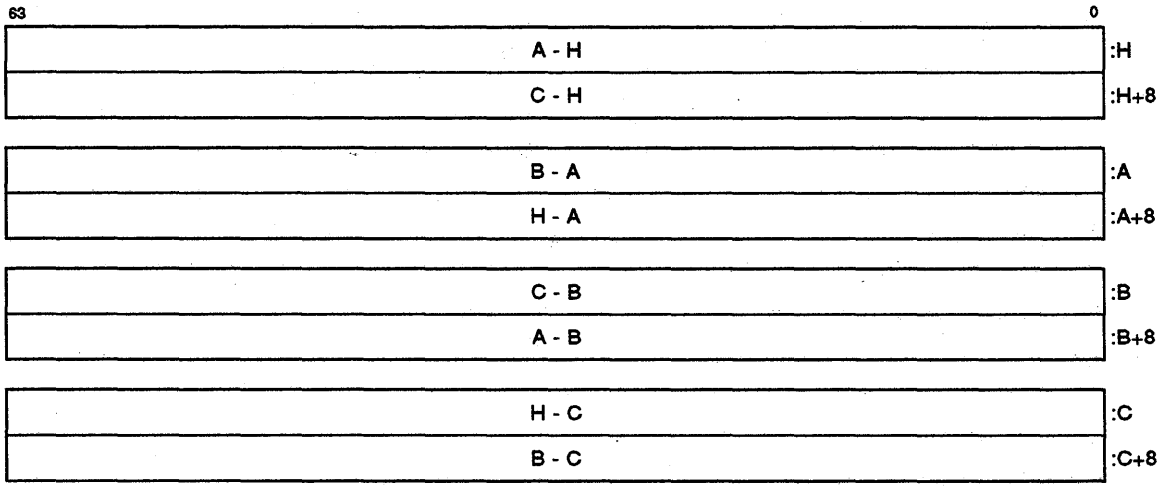


Figure 2–18: Self-Relative Quadword Queue with Three Entries



2.3 Unprivileged OpenVMS Queue PALcode Instructions

The following unprivileged PALcode instructions perform atomic modification of the queue data types that are described in Section 2.2.

Table 2–3: VAX Queue Palcode Instruction Summary

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

2.3.1 Insert Entry into Longword Queue at Head Interlocked

Format:

```
CALL_PAL   INSQHIL                          !PALcode format
```

Operation:

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

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

N <- {retry_amount}                  ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))       ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
      R0 ←  -1, {return}             ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

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

! Check if following addresses can be written
! without causing a memory management exception:
!             entry
!             header + tmp1
IF {all memory accesses can NOT be completed} THEN
   BEGIN                          ! Release secondary interlock.
    (R16) ←  tmp0
    {initiate memory management fault}
   END
```

```
! All accesses can be done so enqueue the entry

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

MB

(R16)<31:0> ←  -tmp2                    ! Forward link of header
                                        ! Release lock
IF tmp1 EQ 0 THEN
  R0 ←  1                               ! Queue was empty
ELSE
  R0 ←  0                               ! Queue was not empty
END
```

Exceptions:

Access Violation

Fault on Read

Fault on Write

Illegal Operand

Translation Not Valid

Instruction Mnemonics:

CALL_PAL INSQHIL Insert into Longword Queue at Head Interlocked

Description:

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

If the entry inserted was the first one in the queue, R0 is set to a 1; else it is set to a 0. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. Before the insertion, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6). If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

2.3.2 Insert Entry into Longword Queue at Head Interlocked Resident

Format:

```
CALL_PAL   INSQHILR                          !PALcode format
```

Operation:

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

N <- {retry_amount}                  ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))       ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
      R0 ←  -1, {return}             ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

MB
(R16)<31:0> ←  -tmp2                    ! Forward link of header
                                       ! Release the lock
IF tmp1 EQ 0 THEN
   R0 ← 1                               ! Queue was empty
ELSE
   R0 ←  0                              ! Queue was not empty
END
```

Exceptions:

Illegal Operand

Instruction Mnemonics:

CALL_PAL	INSQHILR	Insert Entry into Longword Queue at Head Interlocked Resident

Description:

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

If the entry inserted was the first one in the queue, R0 is set to a 1; else it is set to a 0. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

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

2.3.3 Insert Entry into Quadword Queue at Head Interlocked

Format:

```
CALL_PAL   INSQHIQ                          !PALcode format
```

Operation:

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

N <- {retry_amount}*                    ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))          ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN                 ! Try to set secondary interlock.
      R0 ←  -1, {return}                ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

! Check if following addresses can be written
!  without causing a memory management exception:
!            entry
!            header + tmp1
IF {all memory accesses can NOT be completed} THEN
  BEGIN                         ! Release secondary interlock
   (R16) ←  tmp1
   {initiate memory management fault}
  END
```

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

MB

(R16) ←  -tmp2                          ! Forward link of header
                                        ! Release the lock.
IF tmp1 EQ 0 THEN
   R0 ←  1                              ! Queue was empty
ELSE
   R0 ←  0                              ! Queue was not empty
END
```

Exceptions:

Access Violation

Fault on Read

Fault on Write

Illegal Operand

Translation Not Valid

Instruction Mnemonics:

CALL_PAL INSQHIQ Insert into Quadword Queue at Head Interlocked

Description:

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

If the entry inserted was the first one in the queue, R0 is set to a 1; else it is set to a 0. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. Before the insertion, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6). If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

2.3.4 Insert Entry into Quadword Queue at Head Interlocked Resident

Format:

CALL_PAL INSQHIQR !PALcode format

Operation:

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

N <- {retry_amount}                    ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ← (R16))          ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN                ! Try to set secondary interlock.
      R0 ← -1, {return}                ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

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

MB
(R16) ← -tmp2                          ! Forward link of header,
                                       ! Release the lock
IF tmp1 EQ 0 THEN
   R0 ← 1                              ! Queue was empty
ELSE
   R0 ← 0                              ! Queue was not empty
END
```

Exceptions:

Illegal Operand

Instruction Mnemonics:

CALL_PAL	INSQHIQR	Insert Entry into Quadword Queue at Head Interlocked Resident

Description:

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

If the entry inserted was the first one in the queue, R0 is set to a 1; else it is set to a 0. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

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

2.3.5 Insert Entry into Longword Queue at Tail Interlocked

Format:

```
CALL_PAL   INSQTIL                          !PALcode format
```

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!    -1 if the secondary interlock was set
!     0 if the entry was not empty before adding this entry
!     1 if the entry was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
! Header cannot be equal to entry.
!
! check entry and header alignment and
! that the header and entry not same location and
! that the header and entry are valid 32 bit addresses
IF {R16<2:0> NE 0} OR {R17<2:0> NE 0} OR {R16 EQ R17} OR
   {SEXT(R16<31:0>) NE R16} OR {SEXT(R17<31:0>) NE R16} THEN
   BEGIN
    {illegal operand exception}
   END

N <- {retry_amount}                    ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))         ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN                ! Try to set secondary interlock.
      R0 ←  -1, {return}               ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

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

```
! Check if following addresses can be written
!  without causing a memory management exception:
!            entry
!            header + (header + 4)
IF {all memory accesses can NOT be completed} THEN
   BEGIN                              ! Release secondary interlock
    (R16) ←  tmp0
    {initiate memory management fault}
   END

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

Exceptions:

Access Violation

Fault on Read

Fault on Write

Illegal Operand

Translation Not Valid

Instruction Mnemonics:

CALL_PAL INSQTIL Insert into Longword Queue at Tail Interlocked

Description:

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

If the entry inserted was the first one in the queue, R0 is set to a 1; else it is set to a 0. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. Before performing any

part of the operation, the processor validates that the insertion can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6). If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

2.3.6 Insert Entry into Longword Queue at Tail Interlocked Resident

Format:

```
CALL_PAL   INSQTILR                              !PALcode format
```

Operation:

```
! R16 contains the address of the queue header
! R17 contains the address of the new entry
! R0 receives status:
!    -1 if the secondary interlock was set
!     0 if the entry was not empty before adding this entry
!     1 if the entry was empty before adding this entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
! Header cannot be equal to entry.
! All parts of the Queue must be memory resident
N <- {retry_amount}                  ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ← (R16))        ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
      R0 ← -1, {return}              ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ← N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ← -1, {return} ! Retry exceeded

MB

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

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

MB

(R16)<31:0> ← tmp1                   ! Forward link
                                     ! Release the lock
IF tmp1 EQ -tmp3 THEN
  R0 ← 1                             ! Queue was empty
ELSE
  R0 ← 0                             ! Queue was not empty
END
```

Exceptions:

Illegal Operand

Instruction Mnemonics:

CALL_PAL	INSQTILR	Insert Entry into Longword Queue at Tail Interlocked Resident

Description:

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

If the entry inserted was the first one in the queue, R0 is set to a 1; else it is set to a 0. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

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

2.3.7 Insert Entry into Quadword Queue at Tail Interlocked

Format:

```
CALL_PAL   INSQTIQ                              !PALcode format
```

Operation:

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

N <- {retry_amount}                    ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))         ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN                ! Try to set secondary interlock.
      R0 ←  -1, {return}               ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

! Check if following addresses can be written
!  without causing a memory management exception:
!            entry
!            header + (header + 8)
IF {all memory accesses can NOT be completed} THEN
   BEGIN                               ! Release secondary interlock.
    (R16) ←  tmp1
    {initiate memory management fault}
   END
```

```
! All accesses can be done so enqueue the entry
tmp3 ←  R16 - R17
(R17) ←  tmp3                          ! Forward link
(R17 + 8) ←  tmp2 + tmp3               ! Backward link
IF {tmp2 NE 0} THEN                    ! Forward link of predecessor
   (R16+tmp2) ←  -tmp3 - tmp2
ELSE
   tmp1 ←  {-tmp3 - tmp2}
(R16+8) ←  -tmp3                       ! Backward link of header

MB

(R16) ←  tmp1                          ! Forward link
                                       ! Release the lock

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

Exceptions:

Access Violation

Fault on Read

Fault on Write

Illegal Operand

Translation Not Valid

Instruction Mnemonics:

CALL_PAL INSQTIQ Insert into Quadword Queue at Tail Interlocked

Description:

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

If the entry inserted was the first one in the queue, R0 is set to a 1 else it is set to a 0. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. Before performing any part of the operation, the processor validates that the insertion can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6). If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

2.3.8 Insert Entry into Quadword Queue at Tail Interlocked Resident

Format:

```
CALL_PAL   INSQTIQR                          !PALcode format
```

Operation:

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

N <- {retry_amount}                    ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))         ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN                ! Try to set secondary interlock.
      R0 ←  -1, {return}               ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

MB

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

Exceptions:

Illegal Operand

Instruction Mnemonics:

CALL_PAL	INSQTIQR	Insert Entry into Quadword Queue at Tail Interlocked Resident

Description:

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

If the entry inserted was the first one in the queue, R0 is set to a 1 else it is set to a 0. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

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

2.3.9 Insert Entry into Longword Queue

Format:

CALL_PAL INSQUEL !PALcode format

Operation:

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

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

Exceptions:

Access Violation

Fault on Read

Fault on Write

Translation Not Valid

Instruction Mnemonics:

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

Description:

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

In either case, if the entry inserted was the first one in the queue, a 1 is returned in R0; otherwise a 0 is returned in R0. The insertion is a non-interruptible operation. Before performing any part of the insertion, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6).

2.3.10 Insert Entry into Quadword Queue

Format:

```
CALL_PAL   INSQUEQ                          !PALcode format
```

Operation:

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

IF opcode EQ INSQUEQ/D THEN
   IF {r16<3:0> NE 0} THEN
     BEGIN
      {illegal operand exception}
     END
   tmp2 ←  (R16)          ! Address of predecessor
ELSE
   tmp2 ←  R16
END
IF {tmp2<3:0> NE 0} OR {R17<3:0> NE 0} THEN
   BEGIN
    {illegal operand exception}
   END
IF {all memory accesses can be completed} THEN
   BEGIN
    tmp ←  (tmp2)          ! Get forward link of entry
    IF {tmp<3:0> NE 0} THEN
     BEGIN                 ! Check alignment
      {illegal operand exception}
     END
    (R17) ←  tmp           ! Set forward link of entry
    (R17 + 8) ←  tmp2      ! Backward link of entry
    (tmp + 8) ←  R17       ! Backward link of successor
    (tmp2) ←  R17          ! Forward link of predecessor
    IF tmp EQ tmp2 THEN
      R0 ←   1
    ELSE
      R0 ←   0
   END
ELSE
  BEGIN
   {initiate fault}
  END
END
```

Exceptions:

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

Instruction Mnemonics:

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

Description:

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

In either case, if the entry inserted was the first one in the queue, a 1 is returned in R0; otherwise a 0 is returned in R0. The insertion is a non-interruptible operation. Before performing any part of the insertion, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6). R0 is unpredictable if an exception occurs. The relative order of reporting memory management and illegal operand exceptions is unpredictable.

2.3.11 Remove Entry from Longword Queue at Head Interlocked

Format:

```
CALL_PAL   REMQHIL                          !PALcode format
```

Operation:

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

N <- {retry_amount}                ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))     ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN            ! Try to set secondary interlock.
      R0 ←  -1, {return}           ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

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

```
tmp2 ←  SEXT({R16 + tmp1}<31:0>)
IF {tmp1 EQL 0} THEN
  tmp3 ←  R16
ELSE
  tmp3 ←  SEXT({tmp2 + SEXT((tmp2)<31:0>)})
IF tmp3<2:0> NE 0 THEN          ! Check Alignment
  BEGIN                         ! Release secondary interlock
   (R16) ←  tmp0
   {illegal operand exception}
  END
(tmp3 + 4)<31:0> ←  R16 - tmp3 ! Backward link of successor
MB
(R16)<31:0> ←  tmp3 - R16       ! Forward link of header
                                ! Release lock
IF tmp1 EQ 0 THEN
  R0 ←  0                       ! Queue was empty
ELSE
  BEGIN
   IF {tmp3 - R16} EQ 0 THEN
    R0 ←  2                     ! Queue now empty
   ELSE
    R0 ←  1                     ! Queue not empty
  END
END
R1 ←  tmp2                      ! Address of removed entry
```

Exceptions:

Access Violation

Fault on Read

Fault on Write

Illegal Operand

Translation Not Valid

Instruction Mnemonics:

CALL_PAL REMQHIL Remove from Longword Queue at Head Interlocked

Description:

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

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal and the queue is empty after the removal, a 2 is returned

in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6.

2.3.12 Remove Entry from Longword Queue at Head Interlocked Resident

Format:

```
CALL_PAL   REMQHILR                          !PALcode format
```

Operation:

```
! R16 contains the address of the queue header
! R0 receives status:
!        -1 if the secondary interlock was set
!         0 if the queue was empty
!         1 if entry removed and queue still not empty
!         2 if entry removed and queue empty
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
! Header and entries must be quadword aligned.
! All parts of the Queue must be memory resident
N <- {retry_amount}                  ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))       ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
      R0 ←  -1, {return}             ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

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

MB
(R16)<31:0> ←  tmp3 - R16            ! Forward link of header
                                     ! Release lock
IF tmp1 EQ 0 THEN
  R0 ←  0                            ! Queue was empty
ELSE
  BEGIN
   IF {tmp3 - R16} EQ 0 THEN
     R0 ←  2                         ! Queue now empty
   ELSE
     R0 ←  1                         ! Queue not empty
  END
END
R1 ←  tmp2                           ! Address of removed entry
```

Exceptions:

Illegal Operand

Instruction Mnemonics:

CALL_PAL	REMQHILR	Remove Entry from Longword Queue at Head Interlocked Resident

Description:

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

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal and the queue is empty after the removal, a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a non-interruptible operation.

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

2.3.13 Remove Entry from Quadword Queue at Head Interlocked

Format:

```
CALL_PAL   REMQHIQ                          !PALcode format
```

Operation:

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

N <- {retry_amount}                  ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))       ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
      R0 ←  -1, {return}             ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

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

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

IF tmp3<3:0> NE 0 THEN       ! Check Alignment
   BEGIN                     ! Release secondary interlock
    (R16) ←  tmp1
    {illegal operand exception}
   END

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

MB

(R16) ←  tmp3 - R16          ! Forward link of header
                             ! Release lock
IF tmp1 EQ 0 THEN
   R0 ←  0                   ! Queue was empty
ELSE
  BEGIN
    IF {tmp3 - R16} EQ 0 THEN
      R0 ←  2                ! Queue now empty
    ELSE
      R0 ←  1                ! Queue not empty
  END
END
R1 ←  tmp2                   ! Address of removed entry
```

Exceptions:

Access Violation

Fault on Read

Fault on Write

Illegal Operand

Translation Not Valid

Instruction Mnemonics:

CALL_PAL REMQHIQ Remove from Quadword Queue at Head Interlocked

Description:

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

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal a 2 is returned

in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6).

2.3.14 Remove Entry from Quadword Queue at Head Interlocked Resident

Format:

```
CALL_PAL   REMQHIQR                          !PALcode format
```

Operation:

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

N <- {retry_amount}                    ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))         ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN                ! Try to set secondary interlock.
      R0 ←  -1, {return}               ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

MB

(R16) ←  tmp3 - R16                    ! Forward link of header
                                       ! Release lock
IF tmp1 EQ 0 THEN
   R0 ←  0                             ! Queue was empty
ELSE
  IF {tmp3 - R16} EQ 0 THEN
     R0 ←  2                           ! Queue now empty
  ELSE
     R0 ←  1                           ! Queue not empty
END
R1 ←  tmp2                             ! Address of removed entry
```

Exceptions:

Illegal Operand

Instruction Mnemonics:

CALL_PAL REMQHIQR Remove Entry from Quadword Queue at Head Interlocked Resident

Description:

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

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a non-interruptible operation.

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

2.3.15 Remove Entry from Longword Queue at Tail Interlocked

Format:

```
CALL_PAL   REMQTIL                          !PALcode format
```

Operation:

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

N <- {retry_amount}                  ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))       ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
      R0 ←  -1, {return}             ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

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

!Check if the following can be done without
! causing a memory management exception:
!  read contents of header + (header + 4) {if tmp1 NE 0}
!  write into header + (header + 4)
!   + (header + 4 + (header + 4)){if tmp1 NE 0}
IF {all memory accesses can NOT be completed} THEN
  BEGIN                       ! Release secondary interlock
   (R16) ←  tmp0
   {initiate memory management fault}
  END
```

```
addr ←  SEXT( {R16 + tmp5}<31:0> )
tmp2 ←  SEXT( {addr + SEXT( (addr+4)<31:0>)}<31:0> )
IF tmp2<2:0> NE 0 THEN          ! Check alignment
  BEGIN                         ! Release secondary interlock
   (R16) ←  tmp0
   {illegal operand exception}
  END

(R16 + 4)<31:0> ←  tmp2 - R16   ! Backward link of header
IF {tmp2 EQL R16} THEN
  (R16)<31:0> ←  0              ! Forward link, release lock
ELSE
  BEGIN
   (tmp2)<31:0> ←  R16 - tmp2   ! Forward link of predecessor
  MB
  (R16)<31:0> ←  tmp1           ! Release lock
 END
IF tmp1 EQ 0 THEN
  R0 ←  0                       ! Queue was empty
ELSE
  BEGIN
   IF {tmp2 - R16} EQ 0 THEN
     R0 ←  2                    ! Queue now empty
   ELSE
     R0 ←  1                    ! Queue not empty
  END
R1 ←  addr                      ! Address of removed entry
```

Exceptions:

Access Violation

Fault on Read

Fault on Write

Illegal Operand

Translation Not Valid

Instruction Mnemonics:

CALL_PAL REMQTIL Remove from Longword Queue at Tail Interlocked

Description:

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

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal a 2 is returned

in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6).

2.3.16 Remove Entry from Longword Queue at Tail Interlocked Resident

Format:

```
CALL_PAL   REMQTILR                          !PALcode format
```

Operation:

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

N <- {retry_amount}                  ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))       ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
      R0 ←  -1, {return}             ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

tmp1 ←  SEXT(tmp0<31:0>)
tmp5 ←  SEXT(tmp0<63:32>)
addr ←  SEXT( {R16 + tmp5}<31:0> )
tmp2 ←  SEXT( {addr + SEXT( (addr+4)<31:0>)}<31:0> )
(R16 + 4)<31:0> ←  tmp2 - R16        ! Backward link of header
IF {tmp2 EQL R16} THEN
   (R16)<31:0> ←  0                  ! Forward link, release lock
ELSE
  BEGIN
   (tmp2)<31:0> ←  R16 - tmp2        ! Forward link of predecessor
   MB
   (R16)<31:0> ←  tmp1               ! Release lock
  END
  IF tmp1 EQ 0 THEN
    R0 ←  0                          ! Queue was empty
  ELSE
    IF {tmp2 - R16} EQ 0 THEN
        R0 ←  2                      ! Queue now empty
    ELSE
        R0 ←  1                      ! Queue not empty
  END
END
R1 ←  addr                           ! Address of removed entry
```

Exceptions:

Illegal Operand

Instruction Mnemonics:

CALL_PAL	REMQTILR	Remove Entry from Longword Queue at Tail Interlocked Resident

Description:

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

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a non-interruptible operation.

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

2.3.17 Remove Entry from Quadword Queue at Tail Interlocked

Format:

```
CALL_PAL   REMQTIQ                          !PALcode format
```

Operation:

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

N <- {retry_amount}                    ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))         ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN                ! Try to set secondary interlock.
      R0 ←  -1, {return}               ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

tmp5 ←  (R16+8)
IF tmp5<3:0> NE 0 THEN     ! Check Alignment
   BEGIN                   ! Release secondary interlock
    (R16) ←  tmp1
    {illegal operand exception}
   END
! Check if the following can be done without
!  causing a memory management exception:
!   read contents of header + (header + 8) {if tmp1 NE 0}
!   write into header + (header + 8)
!   + (header + 8 + (header + 8)){if tmp1 NE 0}
IF {all memory accesses can NOT be completed} THEN
   BEGIN                   ! Release secondary interlock
    (R16) ←  tmp1
    {initiate memory management fault}
   END
```

```
addr ←  R16 + tmp5
tmp2 ←  addr + (addr + 8)
IF tmp2<3:0> NE 0 THEN      ! Check alignment
   BEGIN                    ! Release secondary interlock
    (R16) ←  tmp1
    {illegal operand exception}
   END

(R16 + 8) ←  tmp2 - R16     ! Backward link of header
IF {tmp2 EQL R16} THEN
   (R16) ←  0               ! Forward link, release lock
ELSE
   BEGIN
    (tmp2) ←  R16 - tmp2    ! Forward link of predecessor
    MB
    (R16) ←  tmp1           ! Release lock
   END
END
IF tmp1 EQ 0 THEN
  R0 ←  0                   ! Queue was empty
ELSE
  BEGIN
   IF {tmp2 - R16} EQ 0 THEN
     R0 ←  2                ! Queue now empty
   ELSE
     R0 ←  1                ! Queue not empty
  END
END
R1 ←  addr                  ! Address of removed entry
```

Exceptions:

Access Violation

Fault on Read

Fault on Write

Illegal Operand

Translation Not Valid

Instruction Mnemonics:

CALL_PAL REMQTIQ Remove from Quadword Queue at Tail Interlocked

Description:

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

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6).

2.3.18 Remove Entry from Quadword Queue at Tail Interlocked Resident

Format:

```
CALL_PAL   REMQTIQR                          !PALcode format
```

Operation:

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

N <- {retry_amount}                  ! Implementation-specific
REPEAT
   LOAD_LOCKED (tmp0 ←  (R16))       ! Acquire hardware interlock.
   IF tmp0<0> EQ 1 THEN              ! Try to set secondary interlock.
      R0 ←  -1, {return}             ! Already set
   done ← STORE_CONDITIONAL ((R16) ← {TMP0 OR R1} )
   N ←  N - 1
UNTIL {done EQ 1} OR {N EQ 0}
IF done NEQ 1, R0 ←  -1, {return} ! Retry exceeded

MB

tmp5 ←  (R16+8)
addr ←  R16 + tmp5
tmp2 ←  addr + (addr + 8)
(R16 + 8) ←  tmp2 - R16                  ! Backward link of header
IF {tmp2 EQL R16} THEN
  (R16) ←  0                             ! Forward link, release lock
ELSE
  BEGIN
   (tmp2) ←  R16 - tmp2                  ! Forward link of predecessor
   MB
   (R16) ←  tmp1                         ! Release lock
  END
END
IF tmp1 EQ 0 THEN
   R0 ←  0                               ! Queue was empty
ELSE
   IF {tmp2 - R16} EQ 0 THEN
      R0 ←  2                            ! Queue now empty
   ELSE
      R0 ←  1                            ! Queue not empty
END
R1 ←  addr                               ! Address of removed entry
```

Exceptions:

Illegal Operand

Instruction Mnemonics:

CALL_PAL	REMQTIQR	Remove Entry from Quadword Queue at Tail Interlocked Resident

Description:

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

If the queue was empty prior to this instruction and secondary interlock succeeded, a 0 is returned in R0. If the interlock succeeded and the queue was not empty at the start of the removal, and the queue is empty after the removal a 2 is returned in R0. If the instruction fails to acquire the secondary interlock after "N" retry attempts, then (in the absence of exceptions) R< 0> is set to a –1. The value "N" is implementation dependent. \ The selected initial value of N is 20.\

The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a non-interruptible operation.

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

2.3.19 Remove Entry from Longword Queue

Format:

```
CALL_PAL   REMQUEL                          !PALcode format
```

Operation:

```
! R16 contains the address of the entry to remove
!     or the address of the 32 bit address of the
!     entry for REMQUEL/D
! R0 receives status:
!       -1 if the queue was empty
!        0 if the queue is empty after removing an entry
!        1 if the queue is not empty after removing an entry
! R1 receives the address of the removed entry
!
! Must have write access to header and queue entries
IF opcode EQ REMQUEL/D THEN
    R1 ←  SEXT((R16)<31:0>)
ELSE
   R1 ←  SEXT(R16<31:0>)

IF {all memory accesses can be completed} THEN
   BEGIN
    tmp1 ←  (R1)<31:0>                  ! Forward Link of Predecessor
    ((R1+4)<31:0>)<31:0> ←  tmp1
    tmp2 ←  (R1+4)<31:0>                ! Backward Link of Successor
    ((R1)<31:0>+4)<31:0> ←  tmp2
    R0 ←  1                             ! Queue not empty
    IF {tmp1 EQ tmp2} THEN
      R0 ←  0                           ! Queue now empty
    IF {R1 EQ tmp2} THEN
      R0 ←  -1                          ! Queue was empty
   END
ELSE
  BEGIN
   {initiate fault}
  END
END
```

Exceptions:

Access Violation

Fault on Read

Fault on Write

Translation Not Valid

Instruction Mnemonics:

CALL_PAL	REMQUEL	Remove Entry from Longword Queue
CALL_PAL	REMQUEL/D	Remove Entry from Longword Queue Deferred

Description:

REMQUEL removes the entry addressed by R16 from the longword absolute queue. The address of the removed entry is returned in R1. REMQUEL/D performs the same operation on the queue entry addressed by the longword addressed by R16.

In either case, if there was no entry in the queue to be removed, R0 is set to –1. If there was an entry to remove and the queue is empty at the end of this instruction, R0 is set to 0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. The removal is a non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6).

2.3.20 Remove Entry from Quadword Queue

Format:

```
CALL_PAL   REMQUEQ                          !PALcode format
```

Operation:

```
! R16 contains the address of the entry to remove
!     or address of address of entry for REMQUEQ/D
! R0 receives status:
!       -1 if the queue was empty
!        0 if the queue is empty after removing an entry
!        1 if the queue is not empty after removing an entry
! R1 receives the address of the removed entry
! Must have write access to header and queue entries
! Header and entries must be octaword aligned
IF opcode EQ REMQUEQ/D THEN
   IF {r16<3:0> NE 0} THEN
     BEGIN
      {illegal operand exception}
     END
   R1 ← (R16)
ELSE
   R1 ← R16
IF {R1<3:0> NE 0} THEN   ! Check alignment
   BEGIN
    {illegal operand exception}
   END
 IF {all memory accesses can be completed} THEN
   BEGIN
    tmp1 ← (R1)         ! Forward link of Predecessor
    IF {tmp1<3:0> NE 0} THEN
     BEGIN              ! Check alignment
      {illegal operand exception}
     END
    tmp2 ← (R1+8)       ! Find predecessor
    IF {tmp2<3:0> NE 0} THEN
     BEGIN              ! Check alignment
      {illegal operand exception}
     END
    (tmp2) ← tmp1       ! Update Forward link of predecessor
    ((R1)+8) ← tmp2
    R0 ← 1              ! Queue not empty
    IF {tmp1 EQ tmp2} THEN
     R0 ← 0             ! Queue now empty
    IF {R1 EQ tmp2} THEN
     R0 ← -1            ! Queue was empty
   END
ELSE
   BEGIN
    {initiate fault}
   END
END
```

Exceptions:

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

Instruction Mnemonics:

CALL_PAL	REMQUEQ	Remove Entry from Quadword Queue
CALL_PAL	REMQUEQ/D	Remove Entry from Quadword Queue Deferred

Description:

REMQUEQ removes the queue entry addressed by R16 from the quadword absolute queue. The address of the removed entry is returned in R1. REMQUEL/D performs the same operation on the queue entry addressed by the quadword addressed by R16.

In either case, if there was no entry in the queue to be removed, R0 is set to –1. If there was an entry to remove and the queue is empty at the end of this instruction, R0 is set to 0. If there was an entry to remove and the queue is not empty at the end of this instruction, R0 is set to 1. The removal is a non-interruptible operation. Before performing any part of the removal, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 3 and 6). R0 and R1 are unpredictable if an exception occurs. The relative order of reporting memory management and illegal operand exceptions is unpredictable.

2.4 Unprivileged VAX Compatibility PALcode Instructions

The Alpha architecture provides the following PALcode instructions for use in translated VAX code. These instructions are not a permanent part of the architecture and will not be available in some future implementations. They are provided to help customers preserve VAX instruction atomicity assumptions in porting code from VAX to Alpha. These calls should be user mode. They must not be used by any code other than that generated by the VEST software translator and its supporting runtime code (TIE).

\ When they are removed from the architecture, it would be good if they trapped in a way that they could be functionally software emulated many years in the future, even if the atomicity is not retained in the software emulation. This would allow very old translated images to run in 1998 and beyond, but perhaps restricted to a single processor and some restriction around AST delivery.

They may be removed and not emulated after the first two full generations of Alpha implementations, that is, about 1995. \

2.4.1 Atomic Move Operation

Format:

AMOVRR	!PALcode format
AMOVRM	!PALcode format

Operation:

```
! R16 contains the first source
! R17 contains the first destination address
! R18 contains the first length
! R19 contains the second source
! R20 contains the second destination address
! R21 contains the second length
CASE
  AMOVRR:
        IF intr_flag EQ 0 THEN
            R18 ← 0
            {return}
        END

        intr_flag ← 0
        (R17) ← R16  ! length specified by R18<1:0>
        (R20) ← R19  ! length specified by R21<1:0>
        IF {both moves successful} THEN
            R18 ← 1
        ELSE
            R18 ← 0
        END

   AMOVRM:
        IF intr_flag EQ 0 THEN
           R18 ← 0
           {return}
        END

        intr_flag ← 0
        (R17) ← R16  ! length specified by R18<1:0>
        IF R21<5:0> NE 0 THEN
            BEGIN
               IF R19<1:0> NE 00 OR R20<1:0> NE 00
                  {Illegal operand exception}
               ELSE
                  (R20) ← (R19) ! length specified by R21<5:0>
               END
        IF {both moves successful} THEN
            R18 ← 1
        ELSE
            R18 ← 0
        END
ENDCASE
```

Exceptions:

AMOVRR:	Access Violation
	Fault On Write
	Translation Not Valid
AMOVRM:	Access Violation
	Fault On Read
	Fault On Write
	Illegal Operand
	Translation Not Valid

Instruction Mnemonics:

CALL_PAL	AMOVRR	Atomic Move Register/Register
CALL_PAL	AMOVRM	Atomic Move Register/Memory

Description:

NOTE

The CALL_PAL AMOVxx instructions are *only* for the support of translated VAX code. They will disappear from the architecture at some time in the future. They must be used *only* in translated VAX code and its support routines (TIE).

CALL_PAL AMOVRR

The CALL_PAL AMOVRR instruction specifies two multiprocessor safe register stores to arbitrary byte addresses. Either both stores are done or neither store is done. R18 is set to one if both stores are done, and zero otherwise. The two source registers are R16 and R19. The two destination byte addresses are in R17 and R20. The two lengths are specified in R18<1:0> and R21<1:0>. The length encoding is: 00 - store byte, 01 - store word, 10 - store longword, 11 - store quadword. The low 1, 2, 4, or 8 bytes of the source register are used, respectively. The unused bytes of the source registers are ignored. The unused bits of the length registers (R18<63:2> and R21<63:2>) should be zero (SBZ).

If, upon entry to the PALcode routine, the intr_flag is clear then the instruction sets R18 to zero and exits, doing no stores. Otherwise, intr_flag is cleared and the PALcode routine proceeds. This is the same per-processor intr_flag used by the RS and RC instructions.

The AMOVRR memory addresses may be unaligned. If either store would result in a Translation Not Valid fault, Fault on Write, or Access Violation fault, neither store is done and the corresponding fault is taken. If both stores would result in faults, it is UNPREDICTABLE which one is taken.

NOTE

A fault does not set R18, since the instruction has not been completed.

If both stores can be completed without faulting, they are both attempted using multiprocessor-safe LDQ_L..STQ_C sequences. If all the sequences store successfully with no interruption, the PALcode routine completes with R18 set to one. Otherwise, the PALcode routine completes with R18 set to zero. In addition, R16, R17, R19, R20 and R21 are UNPREDICTABLE upon return from the PALcode routine, even if an exception has occurred.

If the destinations overlap, the stores must appear be done in the order specified.

CALL_PAL AMOVRM

The CALL_PAL AMOVRM instruction specifies one multiprocessor safe register store to an arbitrary byte address, plus an atomic memory-to-memory move of 0 to 63 aligned longwords. Either the store and the move are both done in their entirety or neither is done. R18 is set to one if both are done, and zero otherwise.

The first source register is R16, the first destination address is in R17, and the first length is in R18. These three are specified exactly as in AMOVRR.

The second source address is in R19, the second destination address is in R20, and the second length is in R21<5:0>. The length is a longword length, in the range 0 to 63 longwords (0 to 252 bytes). The unused bytes of the source register R16 are ignored. The unused bits of the length registers registers (R18<63:2> and R21<63:6>) should be zero (SBZ).

If, upon entry to the PALcode routine, the intr_flag is clear then the instruction sets R18 to zero and exits, doing no stores. Otherwise, intr_flag is cleared and the PALcode routine proceeds. This is the same per-processor intr_flag used by the RS and RC instructions.

The memory address in R17 may be unaligned.

If the length for the move is zero, no move is done, no memory accesses are made via R19 and R20, and no fault checking of these addresses is done. In this case, the move is always considered to have succeeded in determining the setting of R18.

If the length in R21 is non-zero, the two addresses in R19 and R20 must be aligned longword addresses, otherwise an Illegal Operand exception is taken.

If either the store or the move would result in a Translation Not Valid, Fault on Read, Fault on Write, or Access Violation fault, neither is done and the corresponding fault is taken. If both would result in faults, it is UNPREDICTABLE which one is taken.

NOTE

A fault does not set R18, since the instruction has not been completed.

If both the store and the move can be completed without faulting, they are both attempted, using multiprocessor-safe LDQ_L..STQ_C sequences for the store. If

all the operations store successfully with no interruption, the PALcode routine completes with R18 set to one. Otherwise, the PALcode routine completes with R18 set to zero. In addition, R16, R17, R19, R20 and R21 are UNPREDICTABLE upon return from the PALcode routine, even if an exception has occurred.

If the memory fields overlap, the store must appear be done first, followed by the move. The ordering of the reads and writes of the move is unspecified. Thus, if the move destination overlaps the move source, the move results are UNPREDICTABLE.

These instructions contain no implicit MB.

Notes:

- Typical use of these instructions would be a sequence starting with CALL_PAL RS and ending with CALL_PAL AMOVxx, Bxx R18,label. The failure path from the conditional branch would eventually go back to the RS instruction. When such a sequence succeeds, it has done everything from the RS up to and including the CALL_PAL AMOVxx completely with no interrupts or exceptions.
- The CALL_PAL AMOVxx instruction is typically be followed by a conditional branch on R18. If the CALL_PAL AMOVxx is likely to succeed, the conditional branch should be a FORWARD branch on failure (BEQ R18,forward_label) or backward branch on success (BNE R18, backward_label), to match the architected branch-prediction rule.

2.5 Unprivileged PALcode Thread Instructions

The PALcode thread instructions provide support for multithread implementations, which require that a given thread be able to generate a reproducable unique value in a "timely" fashion. This value can then be used to index into a structure or otherwise generate further thread unique data.

The two instructions in Table 2–4 are provided to read and write a process unique value from the process's hardware context.

Table 2–4: Unprivileged PALcode Thread Instructions

Mnemonic	Operation
READ_UNQ	Read unique context
WRITE_UNQ	Write unique Context

The process unique value is stored in the HWPCB at [HWPCB+72] when the process is not active. When the process is active, the process unique value can be cached in hardware internal storage or resident in the HWPCB only.

2.5.1 Read Unique Context

Format:

```
CALL_PAL   READ_UNQ                          !PALcode format
```

Operation:

```
IF {internal storage for process unique context} THEN
   R0 ←  {process unique context}
ELSE
   R0 ←  (HWPCB+72)
```

Exceptions:

None

Instruction Mnemonics:

```
CALL_PAL   READ_UNQ        Read Unique Context
```

Description:

The READ_UNQ instruction causes the hardware process (thread) unique context value to be placed in R0. If this value has not previously been written using a CALL_PAL WRITE_UNQ or stored into the quadword in the HWPCB at [HWPCB+72] while the thread was inactive then the result returned in R0 is UNPREDICTABLE. Implementations can cache this unique context value while the hardware process is active. The unique context may be thought of as a "slow register". Typically, this value will be used by software to establish a unique context for a given thread of execution.

2.5.2 Write Unique Context

Format:

```
CALL_PAL    WRITE_UNQ                          !PALcode format
```

Operation:

```
!R16 contains value to be written to the hardware process
!                unique context

IF {internal storage for process unique context} THEN
   {process unique context} ←  R16
ELSE
   (HWPCB+72) ←  R16
```

Exceptions:

None

Instruction Mnemonics:

```
CALL_PAL    WRITE_UNQ       Write Unique Context
```

Description:

The WRITE_UNQ instruction causes the value of R16 to be stored in internal storage for hardware process (thread) unique context, if implemented, or in the HWPCB at [HWPCB+72], if the internal storage is not implemented. When the process is context switched, SWPCTX ensures this value is stored in the HWPCB at [HWPCB+72]. Implementations can cache this unique context value in internal storage while the hardware process is active. The unique context may be thought of as a "slow register". Typically, this value will be used by software to establish a unique context for a given thread of execution.

2.6 Privileged PALcode Instructions

Privileged instructions can be called in Kernel mode only; otherwise, a privileged instruction exception occurs. The following privileged instructions are provided:

Table 2–5: PALcode Privileged Instructions Summary

Mnemonic	Operation
CFLUSH	Cache flush
DRAINA	Drain aborts See *Common Architecture, Chapter 6*
HALT	Halt processor See *Common Architecture, Chapter 6*
LDQP	Load quadword physical
MFPR	Move from processor register
MTPR	Move to processor register
STQP	Store quadword physical
SWPCTX	Swap privileged context

2.6.1 Cache Flush

Format:

```
CALL_PAL   CFLUSH                                  !PALcode format
```

Operation:

```
! R16 contains the Page Frame Number (PFN)
!       of the page to be flushed

IF  PS<CM> NE 0  THEN
   {privileged instruction exception}

{Flush page out of cache(s)}
```

Exceptions:

Privileged Instruction

Instruction Mnemonics:

```
CALL_PAL   CFLUSH      Cache Flush
```

Description:

The CFLUSH instruction may be used to flush an entire physical page specified by the PFN in R16 from any data caches associated with the current processor. All processors must implement this instruction.

On processors which implement a backup power option which maintains only the contents of memory in the event of a powerfail, this instruction is used by the powerfail interrupt handler to force data written by the handler to the battery backed up main memory. After a CFLUSH, the first subsequent load (on the same processor) to an arbitrary address in the target page is either fetched from physical memory or from the data cache of another processor.

Note that in some multiprocessor systems, CFLUSH is not sufficient to ensure that the data are actually written to memory and not exchanged between processor caches. Additional platform-specific cooperation between the powerfail interrupt handlers executing on each processor may be required.

On systems which implement other backup power options (including none), CFLUSH may return without affecting the data cache contents.

To order CFLUSH properly with respect to preceding writes, an MB instruction is needed before the CFLUSH; to order CFLUSH properly with respect to subsequent reads, an MB instruction is needed after the CFLUSH.

2.6.2 Load Quadword Physical

Format:

```
CALL_PAL   LDQP                                    !PALcode format
```

Operation:

```
! R16 contains the quadword aligned physical address
! R0 receives the data from memory

IF PS<CM> NE 0 THEN
  {Privileged Instruction exception}

R0 ←   (R16) {physical access}
```

Exceptions:

Privileged Instruction

Instruction Mnemonics:

CALL_PAL LDQP Load Quadword Physical

Description:

The LDQP instruction fetches and writes to R0 the quadword aligned memory operand, whose physical address is in R16.

If the operand address in R16 is not quadword aligned, the result is UNPREDICTABLE.

2.6.3 Move From Processor Register

Format:

```
CALL_PAL    MFPR_IPR_Name                    !PALcode format
```

Operation:

```
IF  PS<CM> NE 0  THEN
   {privileged instruction exception}

! R16 may contain an IPR specific source operand
{R0 ←  result of IPR specific function}
```

Exceptions:

Privileged Instruction

Instruction Mnemonics:

CALL_PAL MFPR_xxx Move from Processor Register xxx

Description:

The MFPR_xxx instruction reads the internal processor register specified by the PALcode function field and writes it to R0.

Registers R1, R16, and R17 contain unpredictable results after an MFPR.

See Chapter 5 for a description of each IPR.

2.6.4 Move to Processor Register

Format:

CALL_PAL MTPR_IPR_Name !PALcode format

Operation:

```
IF  PS<CM> NE 0  THEN
  {privileged instruction exception}
! R16 may contain an IPR specific source operand

{R0 ←  result of IPR specific function}
{IPR ←  result of IPR specific function}
```

Exceptions:

Privileged Instruction

Instruction Mnemonics:

CALL_PAL MTPR_xxx Move to Processor Register xxx

Description:

The MTPR_xxx instruction writes the IPR-specific source operands in integer registers R16 and R17 (R17 reserved for future use) to the internal processor register specified by the PALcode function field. The effect of loading a processor register is guaranteed to be active on the next instruction.

Registers R1, R16, and R17 contain unpredictable results after an MTPR. The MTPR may return results in R0. If the specific IPR being accessed does not return results in R0, then R0 contains an unpredictable result after an MTPR.

See Chapter 5 for a description of each IPR.

2.6.5 Store Quadword Physical

Format:

CALL_PAL STQP !PALcode format

Operation:

```
! R16 contains the quadword aligned physical address
! R17 contains the data to be written

IF PS<CM> NE 0 then
  {Privileged Instruction exception}

(R16) ← R17 {physical access}
```

Exceptions:

Privileged Instruction

Instruction Mnemonics:

CALL_PAL STQP Store Quadword Physical

Description:

The STQP instruction writes the quadword contents of R17 to the memory location whose physical address is in R16.

If the operand address in R16 is not quadword aligned, the result is UNPREDICTABLE.

2.6.6 Swap Privileged Context

Format:

```
CALL_PAL   SWPCTX                          !PALcode format
```

Operation:

```
! R16 contains the physical address of the new HWPCB.

! check HWPCB alignment

IF R16<6:0> NE 0 THEN
  {reserved operand exception}
IF {PS<CM> NE 0} THEN
  {privileged instruction exception}

! Store old HWPCB contents

(IPR_PCBB + HWPCB_KSP) ←  SP
IF {internal registers for stack pointers}  THEN
  BEGIN
   (IPR_PCBB + HWPCB_ESP) ←  IPR_ESP
   (IPR_PCBB + HWPCB_SSP) ←  IPR_SSP
   (IPR_PCBB + HWPCB_USP) ←  IPR_USP
  END

IF {internal registers for ASTxx}  THEN
  BEGIN
   (IPR_PCBB + HWPCB_ASTSR) ←  IPR_ASTSR
   (IPR_PCBB + HWPCB_ASTEN) ←  IPR_ASTEN
  END
tmp1 ←  PCC
tmp2 ←  ZEXT(tmp1<31:0>)
tmp3 ←  ZEXT(tmp1<63:32>)
(IPR_PCBB + HWPCB_PCC) ←  {tmp2 + tmp3}<31:0>
IF {internal storage for process unique value} THEN
  BEGIN
   (IPR_PCBB + HWPCB_UNQ) ←  process unique value
  END

! Load new HWPCB contents

IPR_PCBB ←  R16

IF {ASNs not implemented in virtual instruction cache} THEN
   {flush instruction cache}

IF {ASNs not implemented in TB} THEN
   IF {IPR_PTBR NE (IPR_PCBB + HWPCB_PTBR)} THEN
     {invalidate trans. buffer entries with PTE<ASM> EQ 0}
ELSE
   IPR_ASN ←  (IPR_PCBB + HWPCB_ASN)
```

```
SP ←  (IPR_PCBB + HWPCB_KSP)
IF {internal registers for stack pointers} THEN
  BEGIN
   IPR_ESP ←  (IPR_PCBB + HWPCB_ESP)
   IPR_SSP ←  (IPR_PCBB + HWPCB_SSP)
   IPR_USP ←  (IPR_PCBB + HWPCB_USP)
  END

IPR_PTBR  ←  (IPR_PCBB + HWPCB_PTBR)

IF {internal registers for ASTxx}  THEN
 BEGIN
  IPR_ASTSR ←  (IPR_PCBB + HWPCB_ASTSR)
  IPR_ASTEN ←  (IPR_PCBB + HWPCB_ASTEN)
 END

IPR_FEN ←  (IPR_PCBB + HWPCB_FEN)
  tmp4 ←  ZEXT((IPR_PCBB + HWPCB_PCC)<31:0>)
  tmp4 ←  tmp4 - tmp2
  PCC<63:32> ←  tmp4<31:0>
IF {internal storage for process unique value} THEN
  BEGIN
    process unique value ←  (IPR_PCBB + HWPCB_UNQ)
  END
IF {internal storage for Data Alignment trap setting} THEN
  BEGIN
    DAT ←  (IPR_PCBB + HWPCB_DAT)
  END
```

Exceptions:

Reserved Operand

Privileged Instruction

Instruction Mnemonics:

CALL_PAL SWPCTX Swap Privileged Context

Description:

The SWPCTX instruction returns ownership of the current Hardware Privileged Context Block (HWPCB) to the operating system and passes ownership of the new HWPCB to the processor. The HWPCB is described in Chapter 4.

SWPCTX saves the privileged context from the internal processor registers into the HWPCB specified by the physical address in the PCBB internal processor register. It then loads the privileged context from the new HWPCB specified by the physical address in R16. Note that the actual sequence of the save and restore operation is not specified so any overlap of the current and new HWPCB storage areas produces UNDEFINED results.

The privileged context includes the four stack pointers, the Page Table Base Register (PTBR), the Address Space Number (ASN), the AST enable and summary registers, the Floating-point enable register (FEN), the Performance monitor (PME) register, the Data alignment trap (DAT) register, and the process cycle counter (PCC). However, PTBR is never saved in the HWPCB and it is UNPREDICTABLE whether or not ASN is saved. These values cannot be changed for a running process. The process integer and floating registers are saved and restored by the operating system. See Figure 4–1 for the HWPCB format.

Any change to the current HWPCB while the processor has ownership results in UNDEFINED operation. All the values in the current HWPCB can be read through IPRs.

If the HWPCB is read while ownership resides with the processor, it is UNPREDICTABLE whether the original or an updated value of a field is read. The processor is free to update an HWPCB field at any time. The decision as to whether or not a field is updated is made individually for each field.

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

PALcode sets up the PCBB at boot time to point to the HWPCB storage area in the Hardware Restart Parameter Block (HWRPB). \ See *Platform Section, Chapter 3.* \

The operation is UNDEFINED if SWPCTX accesses a non-memory region.

A reference to non-existent memory causes a Machine Check. Unimplemented physical address bits are SBZ. The operation is UNDEFINED if any of these bits are set.

NOTE

Processors may keep a copy of each of the per-process stack pointers in internal registers. In those processors, SWPCTX stores the internal registers into the HWPCB. Processors that do not keep a copy of the stack pointers in internal registers, keep only the stack pointer for the current access mode in SP and switch this with the HWPCB contents whenever the current access mode changes.

2.7 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Changed attempt to acquire secondary lock to retry value
2. Modified RSCC and CFLUSH descriptions
3. Removed DRAINA to common PAL chapter
4. Added ECO #29 GENTRAP
5. Added ECO #27 (octaword aligned queues)
6. Added secondary interlock information
7. Added ECO #31 & #44 (AMOVxx PALcode instructions)
8. Added format editing for instructions
9. Added resident Queue Instructions ECO #28
10. IMB and HALT moved to Common PALcode Section
11. Removed priv inst tests from RSCC (an unpriv instruction)
12. Clean up the format for instructions
13. Converted to SDML
14. Added ECO #21, #23, #26
15. Identify queue type, for Queue instructions
16. Modified REI pseudocode
17. Integrate references for Console ECO #15

Revision 4.0, March 29, 1991

1. Put in ECO for PAL Thread Instructions
2. Put in eco requiring current stack be writable for REI instruction
3. Put in eco requiring REMQUEx/D to return address of removed entry in R1
4. Typos
5. Correct cross reference to section 'Replacement of standard PALcode'
6. Impose uniform usage of CASE pseudocode construct
7. Clarify use of R17 and R0 or MTPR instruction
8. Specify R16 and R17 as integer registers for MTPR instruction
9. Replace occurrences of 'Reserved Operand Exception' with 'Illegal PALcode Operand Trap'
10. Clarify that subsettable unprivileged PAL Instructions can individually either be implemented or cause an Illegal Instruction Trap

11. Change references from 'interrupt' to 'AST' in SWASTEN description, and to 'interrupt or AST' in REI description
12. Add Privileged Instruction exception to those experienced by CFLUSH and DRAINA
13. Correct inconsistent titles for INSQHIQ and INSQTIQ Instructions
14. Tweak MFPR_IPR operation definition
15. Add 'Read System Cycle Counter' PALcode description

Revision 3.0, March 2, 1990

1. Fix Bug in /D version of REMQUEx and INSQUEx
2. Add stack fixup to REI
3. Add Memory Barrier to interlocked queues
4. Add section on replacement of PALcode
5. Add Cflush
6. Rework IFLUSH to IMB
7. Remove PAST
8. Define which PAL may be subsetted

Revision 2.0, October 4, 1989

1. Remove test and set/clear interlocked
2. Add deferred addressing to the absolute queues
3. Add drain aborts (DRAINA)
4. Add poll AST (PAST)
5. Remove read/write of inexact exception enable
6. Add CC and FEN to SWPCTX
7. Rework interlocked queues for LDQ/L and STQ/C

Revision 1.0, May 23, 1989

1. First Full Version

Revision 0.0, March 15, 1989

1. Initial Version

Chapter 3

OpenVMS Memory Management (II)

3.1 Introduction

Memory management consists of the hardware and software which control the allocation and use of physical memory. Typically, in a multiprogramming system, several processes may reside in physical memory at the same time; see Chapter 4. OpenVMS Alpha uses memory protection and multiple address spaces to ensure that one process will not affect either other processes or the operating system.

To improve further software reliability, four hierarchical access modes provide memory access control. They are, from most to least privileged: kernel, executive, supervisor, and user. Protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write for each of the four access modes. Accessible pages can be restricted to have only data or instruction access.

A program uses virtual addresses to access its data and instructions. However, before these virtual addresses can be used to access memory, they must be translated into physical addresses. Memory management software maintains tables of mapping information (page tables) that keep track of where each virtual page is located in physical memory. The processor utilizes this mapping information when it translates virtual addresses to physical addresses.

Therefore, memory management provides both memory protection and memory mapping mechanisms. The OpenVMS Alpha memory management architecture is designed to meet several goals:

- Provide a large address space for instructions and data.
- Allow programs to run on hardware with physical memory smaller than the virtual memory used.
- Provide convenient and efficient sharing of instructions and data.
- Allow sparse use of a large address space without excessive page table overhead.
- Contribute to software reliability.
- Provide independent read and write access protection.

3.2 Virtual Address Space

A virtual address is a 64-bit unsigned integer specifying a byte location within the virtual address space. Implementations subset the address space supported to one of four sizes (43, 47, 51, or 55 bits) as a function of page size. The minimal virtual

address size supported is 43 bits. If an implementation supports less than 64-bit virtual addresses it must check that all the VA<63:VA_SIZE> bits are equal to VA<VA_SIZE-1>. This gives two disjoint ranges for valid virtual addresses. For example, for a 43-bit virtual address space valid virtual addresses ranges are 0..3FF FFFF $FFFF_{16}$ and FFFF FC00 0000 0000_{16}..FFFF FFFF FFFF $FFFF_{16}$. Accesses to virtual addresses outside of the valid virtual address ranges for an implementation cause an access violation exception.

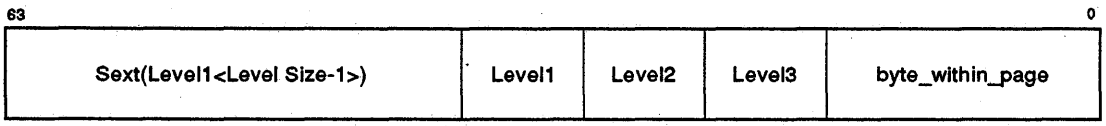
The virtual address space is broken into pages, which are the units of relocation, sharing, and protection. The page size ranges from 8K bytes to 64K bytes. System software should, therefore, allocate regions with differing protection on 64-Kbyte virtual address boundaries to ensure image compatibility across all Alpha implementations.

Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. The operating system controls the virtual-to-physical address mapping tables, and saves the inactive parts of the virtual address space on external storage media.

3.2.1 Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. The virtual address consists of three level-number fields, and a byte_within_page field.

Figure 3–1: Virtual Address Format



The byte_within_page field can be either 13, 14, 15, or 16 bits depending on a particular implementation. Thus, the allowable page sizes are 8K bytes, 16K bytes, 32K bytes, and 64K bytes. Each level-number field contains 0-n bits, where *n* is, for example, 9 with an 8K-byte page size. The level-number fields are the same size for a given implementation.

The level number fields are a function of the page size; all page table entries at any given level do not exceed one page. The PFN field in the PTE is always 32 bits wide. Thus, as the page size grows the virtual and physical address size also grows.

Table 3–1: Virtual Address Options

Page Size (bytes)	Byte Offset (bits)	Level Size (bits)	Virtual Address (bits)	Physical Address (bits)
8 K	13	10	43	45

Table 3–1 (Cont.): Virtual Address Options

Page Size (bytes)	Byte Offset (bits)	Level Size (bits)	Virtual Address (bits)	Physical Address (bits)
16 K	14	11	47	46
32 K	15	12	51	47
64 K	16	13	55	48

3.3 Physical Address Space

Physical addresses are at most 48 bits. A processor may choose to implement a smaller physical address space by not implementing some number of high order bits. The two most significant implemented physical address bits select a caching policy or implementation dependent type of address space. Implementations will use these bits as appropriate for their systems. For example, in a workstation with a 30-bit physical address space, bit <29> might select between memory and non-memory like regions, and bit <28> could enable or disable caching; see *Common Architecture, Chapter 5*.

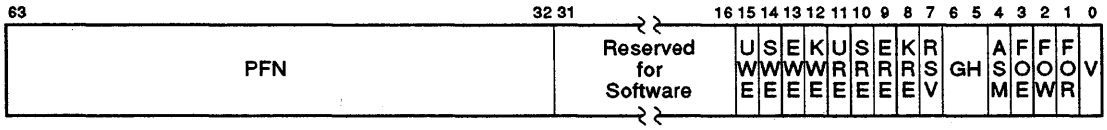
3.4 Memory Management Control

Memory management is always enabled. Implementations must provide an environment for PALcode to service exceptions and to initialize and boot the processor. For example PALcode might run with I-stream mapping disabled and use the privileged CALL_PAL LDQP and STQP instructions to access data stored in physical addresses.

3.5 Page Table Entries

The processor uses a quadword Page Table Entry (PTE) to translate virtual addresses to physical addresses. A PTE contains hardware and software control information and the physical Page Frame Number.

Figure 3–2: Page Table Entry



Fields in the page table entry are interpreted as shown in Table 3–2.

Table 3–2: Page Table Entry

Bits	Description
0	Valid (V) Indicates the validity of the the PFN field. When V is set the PFN field is valid for use by hardware. When V is clear, the PFN field is reserved for use by software. The V bit does not affect the validity of PTE<15:1> bits.
1	Fault On Read (FOR) When set, a Fault On Read exception occurs on an attempt to read any location in the page.
2	Fault On Write (FOW) When set, a Fault On Write exception occurs on an attempt to write any location in the page.
3	Fault On Execute (FOE) When set, a Fault On Execute exception occurs on an attempt to execute an instruction in the page.
4	Address Space Match (ASM) When set, this PTE matches all Address Space Numbers. For a given VA, ASM must be set consistently in all processes, otherwise the address mapping is UNPREDICTABLE.

Table 3–2 (Cont.): Page Table Entry

Bits	Description
6:5	Granularity hint (GH) Software may set these bits to a non-zero value to supply a hint to translation buffer implementations that a block of pages can be treated as a single larger page: 1. The block is an aligned group of 8**N pages, where N is the value of PTE<6:5>, e.g. a group of 1, 8, 64, or 512 pages starting at a virtual address with page_size + 3*N low-order zeros. 2. The block is a group of physically contiguous pages that are aligned both virtually and physically. Within the block, the low 3*N bits of the PFNs describe the identity mapping and the high 32-3*N PFN bits are all equal. 3. Within the block, all PTEs have the same values for bits <15:0>, i.e. protection, fault, granularity, and valid bits. Hardware may use this hint to map the entire block with a single TB entry, instead of 8, 64, or 512 separate TB entries. Note that it is UNPREDICTABLE which PTE values within the block are used if the granularity bits are set inconsistently. **PROGRAMMING NOTE** A granularity hint might be appropriate for a large memory structure such as a frame buffer or nonpaged pool that in fact is mapped into contiguous virtual pages with identical protection, fault, and valid bits.
7	Reserved for future use by DIGITAL. **PROGRAMMING NOTE** The reserved bit will be used by future hardware systems and should not be used by software even if PTE<V> is clear.
8	Kernel Read Enable (KRE) This bit enables reads from kernel mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V=0.
9	Executive Read Enable (ERE) This bit enables reads from executive mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in executive mode, an Access Violation occurs. This bit is valid even when V=0.

Table 3–2 (Cont.): Page Table Entry

Bits	Description
10	Supervisor Read Enable (SRE) This bit enables reads from supervisor mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in supervisor mode, an Access Violation occurs. This bit is valid even when V=0.
11	User Read Enable (URE) This bit enables reads from user mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in user mode, an Access Violation occurs. This bit is valid even when V=0.
12	Kernel Write Enable (KWE) This bit enables writes from kernel mode. If this bit is a 0 and a STORE is attempted while in kernel mode, an Access Violation occurs. This bit is valid even when V=0.
13	Executive Write Enable (EWE) This bit enables writes from executive mode. If this bit is a 0 and a STORE is attempted while in executive mode, an Access Violation occurs. This bit is valid even when V=0.
14	Supervisor Write Enable (SWE) This bit enables writes from supervisor mode. If this bit is a 0 and a STORE is attempted while in supervisor mode, an Access Violation occurs. This bit is valid even when V=0.
15	User Write Enable (UWE) This bit enables writes from user mode. If this bit is a 0 and a STORE is attempted while in user mode, an Access Violation occurs. This bit is valid even when V=0. **NOTE** If a write enable bit is set and the corresponding read enable bit is not, the operation of the processor is UNDEFINED.
31:16	Reserved for software.
63:32	Page Frame Number (PFN) The PFN field always points to a page boundary. If V is set, the PFN is concatenated with the byte_within_page bits of the virtual address to obtain the physical address; see Section 3.7. If V is clear, this field may be used by software.

3.5.1 Changes to Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, the operating system may set or clear the valid bit, change the PFN field as pages are moved to and from external storage media, or modify the software bits. The processor hardware never changes PTEs.

Software must guarantee that each PTE is always consistent within itself. Changing a PTE one field at a time may give incorrect system operation, e.g., setting PTE<V> with one instruction before establishing PTE<PFN> with another. Execution of an interrupt service routine between the two instructions could use an address that would map using the inconsistent PTE. Software can solve this problem by building a complete new PTE in a register and then moving the new PTE to the page table using a Store Quadword instruction (STQ).

Multiprocessing makes the problem more complicated. Another processor could be reading (or even changing) the same PTE that the first processor is changing. Such concurrent access must produce consistent results. Software must use some form of software synchronization to modify PTEs that are already valid. Once a processor has modified a valid PTE, it is possible that other processors in a multiprocessor system may have old copies of that PTE in their Translation Buffer. Software must inform other processors of changes to PTEs.

Software may write new values into invalid PTEs using quadword store instructions (i.e., STQ). Hardware must ensure that aligned quadword reads and writes are atomic operations. The following procedure must be used to change any of the PTE bits <15:0> of a shared valid PTE (PTE<0>=1) such that an access that was allowed before the change is not allowed after the change.

1. The PTE<0> is cleared without changing any of the PTE bits <63:32> and <15:1>.
2. All processors do a TBIS for the VA mapped by the PTE that changed. The VA used in the TBIS must assume that the PTE Granularity hint bits are zero.
3. After all processors have done the TBIS, the new PTE may be written changing any or all fields.

PROGRAMMING NOTE

The procedure above allows the QUEUE instructions that have probed to check that all can complete, to service a TB miss. The QUEUE instruction will use the PTE even though the V bit is clear, if during its initial probe flow the V bit was set.

3.6 Memory Protection

Memory protection is the function of validating whether a particular type of access is allowed to a specific page from a particular access mode. Access to each page is controlled by a protection code that specifies, for each access mode, whether read or write references are allowed.

The processor uses the following to determine whether an intended access is allowed:

- The virtual address, which is used to index page tables.
- The intended access type (read data, write data, or instruction fetch).
- The current access mode from the Processor Status.

If the access is allowed and the address can be mapped (the Page Table Entry is valid), the result is the physical address corresponding to the specified virtual address.

For protection checks, the intended access is read for data loads and instruction fetch, and write for data stores.

If an operand is an address operand, then no reference is made to memory. Hence, the page need not be accessible nor map to a physical page.

3.6.1 Processor Access Modes

There are four processor modes:

- Kernel
- Executive
- Supervisor
- User

The access mode of a running process is stored in the Current Mode bits of the Processor Status (PS); see Section 6.2.

3.6.2 Protection Code

Every page in the virtual address space is protected according to its use. A program may be prevented from reading or writing portions of its address space. Associated with each page is a protection code that describes the accessibility of the page for each processor mode. The code allows a choice of read or write protection for each processor mode.

- Each mode's access can be read/write, read-only, or no-access.
- Read and write accessibility are specified independently.
- The protection of each mode can be specified independently.

The protection code is specified by 8 bits in the PTE; see Table 3–2.

The OpenVMS Alpha architecture allows a page to be designated as execute only by setting the read enable bit for the access mode and by setting the fault on read and write bits in the PTE.

3.6.3 Access Violation Fault

An Access Violation fault occurs if an illegal access is attempted, as determined by the current processor mode and the page's protection field.

3.7 Address Translation

The page tables can be accessed from physical memory, or (to reduce overhead) through a mapping to a linear region of the virtual address space. All implementations must support the virtual access method and are expected to use it as the primary access method to enhance performance.

The following sections describe both access methods.

3.7.1 Physical Access for Page Table Entries

Physical address translation is performed by accessing entries in a three-level page table structure. The Page Table Base Register (PTBR) contains the physical Page Frame Number of the highest level (Level 1) page table. Bits <level1> of the virtual address are used to index into the first level page table to obtain the physical page frame number of the base of the second level (Level 2) page table. Bits <level2> of the virtual address are used to index into the second level page table to obtain the physical page frame number of the base of the third level (Level 3) page table. Bits <level3> of the virtual address are used to index the third level page table to obtain the physical Page Frame Number (PFN) of the page being referenced. The PFN is concatenated with virtual address bits <byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table resides in I/O space, or in nonexistent memory, the operation of the processor is UNDEFINED.

If the first-level or second-level PTE is valid, the protection bits are ignored; the protection code in the third-level PTE is used to determine accessibility. If a first-level or second-level PTE is invalid, an Access Violation occurs if the PTE<KRE> equals zero. An Access Violation on a first-level or second-level PTE implies that all lower-level page tables mapped by that PTE do not exist.

PROGRAMMING NOTE

This mapping scheme does not require multiple contiguous physical pages. There are no length registers. With a page size of 8K bytes, 3 pages (24K bytes) map 8M bytes of virtual address space; 1026 pages (approximately 8M bytes) map an 8-Gbyte address space; and 1,049,601 pages (approximately 8G bytes) map the entire 8T byte 2**43 byte address space.

The algorithm to generate a physical address from a virtual address follows:

```
IF {SEXT(VA<63:VA_SIZE>) NEQ SEXT(VA<VA_SIZE-1>} THEN
   {initiate Access Violation fault}

! Read Physical

level1_pte ←  ({PTBR * page_size} + {8 * VA<level1_number>})

IF level1_pte<V> EQ 0 THEN
   IF level1_pte<KRE> EQ 0 THEN
      {initiate Access Violation fault}
   ELSE
      {initiate Translation Not Valid fault}

! Read Physical

level2_pte ←
   ({level1_pte<PFN> * page_size} + {8 * VA<level2_number>})
```

```
IF level2_pte<V> EQ 0 THEN
   IF level2_pte<KRE> EQ 0 THEN
      {initiate Access Violation fault}
   ELSE
      {initiate Translation Not Valid fault}

! Read Physical

level3_pte ←
   ({level2_pte<PFN> * page_size} + {8 * VA<level3_number>})

IF {{{level3_pte<UWE> EQ 0} AND {write access} AND {PS<CM> EQ 3}} OR
    {{level3_pte<URE> EQ 0} AND {read  access} AND {PS<CM> EQ 3}} OR
    {{level3_pte<SWE> EQ 0} AND {write access} AND {PS<CM> EQ 2}} OR
    {{level3_pte<SRE> EQ 0} AND {read  access} AND {PS<CM> EQ 2}} OR
    {{level3_pte<EWE> EQ 0} AND {write access} AND {PS<CM> EQ 1}} OR
    {{level3_pte<ERE> EQ 0} AND {read  access} AND {PS<CM> EQ 1}} OR
    {{level3_pte<KWE> EQ 0} AND {write access} AND {PS<CM> EQ 0}} OR
    {{level3_pte<KRE> EQ 0} AND {read  access} AND {PS<CM> EQ 0}}}
THEN
   {initiate Access Violation fault}
ELSE
   IF level3_pte<V> EQ 0 THEN
       {initiate Translation Not Valid fault}

IF {level3_pte<FOW> EQ 1} AND { write   access} THEN
   {initiate Fault On Write fault}
IF {level3_pte<FOR> EQ 1} AND { read    access} THEN
   {initiate Fault On Read fault}
IF {level3_pte<FOE> EQ 1} AND { execute access} THEN
   {initiate Fault On Execute fault}

Physical_Address ←
   {level3_pte<PFN> * page_size} OR VA<byte_within_page>
```

3.7.2 Virtual Access for Page Table Entries

To reduce the overhead associated with the address translation in a three-level page table structure, the page tables are mapped into a linear region of the virtual address space. The virtual address of the base of the page table structure is set on a system wide basis and is contained in the VPTB IPR.

When a native mode DTB or ITB Miss occurs, the TBMISS flows attempt to load the level three page table entry using a single virtual mode load instruction.

The algorithm involving the manipulation of the missing VA is:

```
tmp ←  left_shift(VA, {64 - {{lg(PageSize) *4} -9 }} )
tmp ←
    right_shift(tmp,{64 - {{lg(PageSize)*4} -9} + lg(PageSize) -3})
tmp ←  VPTB OR tmp
tmp<2:0> ←   0
```

At this point, tmp contains the VA of the level 3 page table entry. A LDQ from that VA will result in the acquistion of the PTE needed to satisfy the initial TBMISS condition.

However, in the PALcode environment, if a TBMISS occurs during an attempt to fetch the level3 PTE, then it is necessary to use the longer sequence of three dependent loads described in Section 3.7.

Chapter 5 contains the description of the VPTB IPR used to contain the virtual address of the base of the page table structure.

The mapping of the page tables necessary for the correct function of the algorithm is done as follows:

1. Select a $2^{(3*lg(page\_size/8))+3}$ byte-aligned region (an address with 3*lg(page_size /8)+3 low order zeros) in the virtual address space. This value will be written into the VPTB register.
2. Create a level1 PTE to map the page tables as follows:

```
Level1_PTE          ←  0      ! Init all fields to 0
Level1_PTE<63:32>   ←  PFN of Level1 Pagetable
                           ! Set PFN to PFN of level1 pagetable
Level1_PTE<8>       ←  1      ! Kernel Read Enable  (KRE)
Level1_PTE<0>       ←  1      ! Valid bit
```

3. Write the created level1 PTE into the Level1 page table entry that corresponds to the VPTB value.
4. Set all Level1 and Level2 Valid PTEs to allow kernel read access.
5. Write the VPTB register with the selected base value.

NOTE

No validity checks need be made on the value stored in the VPTB in a running system. Therefore, if the VPTB contains an invalid address, the operation is UNDEFINED.

3.8 Translation Buffer

In order to save actual memory references when repeatedly referencing the same pages, hardware implementations include a translation buffer to remember successful virtual address translations and page states.

When the process context is changed, a new value is loaded into the Address Space Number (ASN) internal processor register with a Swap Privileged Context instruction (CALL_PAL SWPCTX); see Section 2.6 and Chapter 4. This causes address translations for pages with PTE<ASM> clear to be invalidated on a processor that does not implement address space numbers. Additionally, when the software changes any part (except for the Software field) of a valid Page Table Entry, it must also move a virtual address within the corresponding page to the Translation Buffer Invalidate Single (TBIS) internal processor register with the MTPR instruction; see Chapter 5.

IMPLEMENTATION NOTE

Some implementations may invalidate the entire Translation Buffer on an MTPR to TBIS. In general, implementations may invalidate more than the required translations in the TB.

The entire Translation Buffer can be invalidated by doing a write to Translation Buffer Invalidate All register (CALL_PAL MTPR_TBIA), and all ASM=0 entries can be invalidated by doing a write to Translation Buffer Invalidate All Process register (CALL_PAL MTPR_TBIAP); see Chapter 5.

The Translation Buffer must not store invalid PTEs. Therefore, the software is not required to invalidate Translation Buffer entries when making changes for PTEs that are already invalid.

The TBCHK internal processor register is available for interrogating the presence of a valid translation in the Translation Buffer; see Chapter 5.

IMPLEMENTATION NOTE

Hardware implementors should be aware that a single, direct mapped TB has a potential problem when a load /store instruction and its data map to the same TB location. If TB misses are handled in PALcode, there could be an endless loop unless the instruction is held in an instruction buffer or a translated physical PC is maintained by the hardware.

3.9 Address Space Numbers

The Alpha architecture allows a processor to optionally implement address space numbers (process tags) to reduce the need for invalidation of cached address translations for process specific addresses when a context switch occurs. The supported ASN range is 0..MAX_ASN. \ MAX_ASN is provided in the HWRPB MAX_ASN field; see *Platform Section, Chapter 3* for a detailed description of the HWRPB. \

NOTE

If an ASN outside of the range 0..MAX_ASN is assigned to a process, the operation of the processor is UNDEFINED.

The address space number for the current process is loaded by software in the Address Space Number (ASN) internal processor register with a Swap Privileged Context instruction. ASNs are processor specific and the hardware makes no attempt to maintain coherency across multiple processors. In a multiprocessor system, software is responsible for ensuring the consistency of TB entries for processes that might be rescheduled on different processors.

\ Systems that support ASNs should have MAX_ASN in the range 13..65535. The number of ASNs should be determined by the market a system is targeting. \

PROGRAMMING NOTE

System software should not assume that the number of ASNs is a power of two. This allows, for example, hardware to use N TB tag bits to encode (2**N)–3 ASN values, one value for ASM=1 PTEs, and one for invalid.

There are several possible ways of using ASNs. There are several complications in a multiprocessor system. Consider the case where a process that executed on processor–1 is rescheduled on processor–2. If a page is deleted or its protection is changed, the TB in processor–1 has stale data. One solution would be to send an interprocessor interrupt to all the processors on which this process could have run and cause them to invalidate the changed PTE. This results in significant overhead in a system with several processors. Another solution would be to have software invalidate all TB entries for a process on a new processor before it can begin execution, if the process executed on another processor during its previous execution. This ensures the deletion of possibly stale TB entries on the new processor. A third solution would assign a new ASN whenever a process is run on a processor that is not the same as the last processor on which it ran.

3.10 Memory Management Faults

Five types of faults are associated with memory access and protection:

- Access Control Violation (ACV)

  Taken when the protection field of the third-level PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode. An Access Control Violation fault is also taken if the KRE bit is zero in an invalid first or second level PTE.

- Fault On Read (FOR)

  Occurs when a read is attempted with PTE<FOR> set.

- Fault On Write (FOW)

  Occurs when a write is attempted with PTE<FOW> set.

- Fault On Execute (FOE)

  Occurs when instruction execution is attempted with PTE<FOE> set.

- Translation Not Valid (TNV)

  Taken when a read or write reference is attempted through an invalid PTE in a first-, second-, or third-level page table.

See Chapter 6 for a detailed description of these faults.

Note that these five faults have distinct vectors in the System Control Block. The Access Violation (ACV) fault takes precedence over the faults TNV, FOR, FOW, and FOE. The Translation Not Valid (TNV) fault takes precedence over the faults FOR, FOW, and FOE.

The faults FOR and FOW can occur simultaneously in the CALL_PAL queue instructions, in which case the order that the exceptions are taken is UNPREDICTABLE; see Section 2.1.

3.11 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Added spacing to code_examples
2. Term level replaces seg in address translation sect
3. Added ECO #17, address translation performance enhancements
4. Converted to SDML
5. Integrate references for Console ECO #15

Revision 4.0, March 29, 1991

1. Typos
2. Clarify reference to TNV and FOx as mutually exclusive
3. Expand on reference to simultaneous occurrence of FOR and FOW in section 'Memory Management Faults'

Revision 3.0, Mar 2, 1990

1. Change ASN to variable size
2. Remove Huge pages and add Granularity hint
3. Add rule on changing PTEs from valid to invalid

Revision 2.0, October 4, 1989

1. Remove references to buffer space
2. Add note that PTE<6:7> are not to be used by software
3. Change name of large pages to huge pages.
4. Add implementation dependent use of high order PFN bits to specify caching policy.

Revision 1.0, May 23, 1989

1. First review distribution.

\

Chapter 4

OpenVMS Process Structure (II)

4.1 Process Definition

A process is the basic entity that is scheduled for execution by the processor. A process represents a single thread of execution and consists of an address space and both hardware and software context.

The hardware context of a process is defined by:

- 31 Integer registers and 31 Floating-point registers
- Processor Status (PS)
- Program Counter (PC)
- 4 stack pointers
- Asynchronous System Trap Enable and summary registers (ASTEN, ASTSR)
- Process Page Table Base Register (PTBR)
- Address Space Number (ASN)
- Floating Enable Register (FEN)
- Process Cycle counter (PCC)
- Process Unique value
- Data Alignment Trap (DAT)
- Performance Monitoring Enable Register (PME)

The software context of a process is defined by operating system software and is system dependent.

A process may share the same address space with other processes or have an address space of its own. There is, however, no separate address space for system software, and therefore, the operating system must be mapped into the address space of each process; see Chapter 3.

In order for a process to execute, its hardware context must be loaded into the integer registers, Floating-point registers, and internal processor registers. While a process is executing, its hardware context is continuously updated. When a process is not being executed, its hardware context is stored in memory.

Saving the hardware context of the current process in memory, followed by loading the hardware context for a new process, is termed context switching. Context

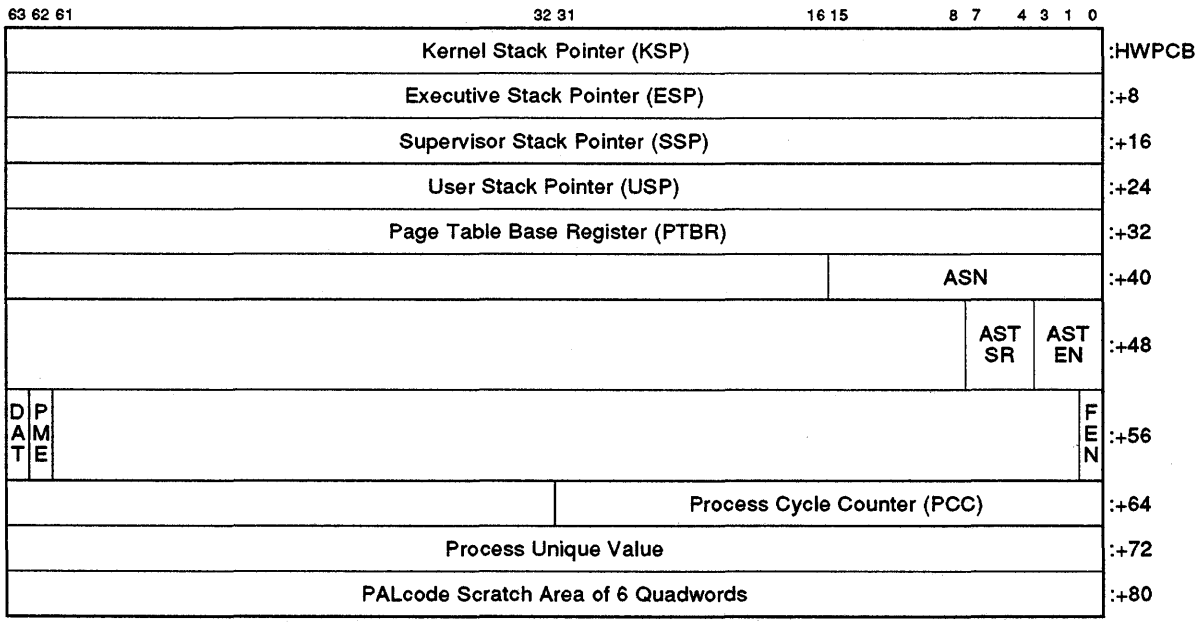
switching occurs as one process after another is scheduled by the operating system for execution.

4.2 Hardware Privileged Process Context

The hardware context of a process is defined by a privileged part which is context switched with the Swap Privileged Context instruction (SWPCTX) (see Section 2.6), and a non-privileged part which is context switched by operating system software.

When a process is not executing, its privileged context is stored in a 128 byte naturally aligned memory structure called the Hardware Privileged Context Block (HWPCB).

Figure 4–1: Hardware Privileged Context Block



The Hardware Privileged Context Block (HWPCB) for the current process is specified by the Privileged Context Block Base register (PCBB); see Chapter 5.

The Swap Privileged Context instruction (SWPCTX) saves the privileged context of the current process into the HWPCB specified by PCBB, loads a new value into PCBB, and then loads the privileged context of the new process into the appropriate hardware registers.

The new value loaded into PCBB, as well as the contents of the Privileged Context Block, must satisfy certain constraints or an UNDEFINED operation results:

1. The physical address loaded into PCBB must be 128 byte aligned and describes sixteen contiguous quadwords that are in a memory-like region; see *Common Architecture, Chapter 5*.
2. The value of PTBR must be the Page Frame Number of an existent page that is in a memory-like region.

It is the responsibility of the operating system to save and load the non-privileged part of the hardware context.

The SWPCTX instruction returns ownership of the current HWPCB to operating system software and passes ownership of the new HWPCB from the operating system to the processor. Any attempt to write a HWPCB while ownership resides with the processor has UNDEFINED results. If the HWPCB is read while ownership resides with the processor, it is UNPREDICTABLE whether the original or an updated value of a field is read. The processor is free to update an HWPCB field at any time. The decision as to whether or not a field is updated is made individually for each field.

If ASNs are not implemented, the ASN field is not read or written by PALcode.

The FEN bit reflects the setting of the FEN IPR.

The DAT bit controls whether data alignment traps that are fixed up in PALcode are reported to the operating system. If the bit is clear, the trap is reported. If the bit is set, after the fixup, return is to the user. See Section 6.6.

Setting the PME bit alerts any performance hardware or software in the system to monitor the performance of this process.

The Process Unique value is that value used in support of multithread implementations. The value is stored in the HWPCB when the process is not active. When the process is active, the value may be cached in hardware internal storage or kept in the HWPCB only.

4.3 Asynchronous System Traps (AST)

Asynchronous System Traps (ASTs) are a means of notifying a process of events that are not synchronized with its execution but which must be dealt with in the context of the process with minimum delay.

Asynchronous System Traps (ASTs) interrupt process execution and are controlled by the AST Enable (ASTEN) and AST Summary (ASTSR) internal processor registers; see Chapter 5.

The AST Enable register (ASTEN) contains an enable bit for each of the four processor access modes. When the bit corresponding to an access mode is set, ASTs for that mode are enabled. The AST enable bit for an access mode may be changed by executing a Swap AST Enable instruction (SWASTEN; see Section 2.6), or by executing a Move To Processor Register instruction specifying ASTEN (MTPR ASTEN; see Chapter 5).

The AST Summary Register (ASTSR) contains a pending bit for each of the four processor access modes. When the bit corresponding to an access mode is set, an AST is pending for that mode.

Kernel mode software may request an AST for a particular access mode by executing a Move To Processor Register instruction specifying ASTSR (MTPR ASTSR); see Chapter 5).

Hardware or PALcode monitors the state of ASTEN, ASTSR, PS<CM>, and PS<IPL>. If PS<IPL> is less than 2, and there is an AST pending and enabled for an access mode that is less than or equal to PS<CM> (i.e. an equal or more privileged access mode), an AST is initiated at IPL 2.

ASTs that are pending and enabled for a less privileged access mode are not allowed to interrupt execution in a more privileged access mode.

4.4 Process Context Switching

Process context switching occurs as one process after another is scheduled for execution by operating system software. Context switching requires the hardware context of one process to be saved in memory followed by the loading of the hardware context for another process into the hardware registers.

The privileged hardware context is swapped with the CALL_PAL Swap Privileged Context instruction (SWPCTX). Other hardware context must be saved and restored by operating system software.

The sequence in which process context is changed is important since the SWPCTX instruction changes the environment in which the context switching software itself is executing. Also, although not enforced by hardware, it is advisable to execute the actual context switching software in an environment which cannot be context switched (i.e. at an IPL high enough that rescheduling cannot occur).

The SWPCTX instruction is the only method provided for loading certain internal processor registers. The SWPCTX instruction always saves the privileged context of the old process and loads the privileged context of a new process. Therefore, a valid HWPCB must be available to save the privileged context of the old process as well as load the privileged context of the new process. \

At system initialization, a valid HWPCB is constructed in the Hardware Restart Parameter Block (HWRPB) for the primary processor; see *Platform Section, Chapter 3*. Thereafter, it is the responsibility of operating system software to ensure a valid HWPCB when executing a SWPCTX instruction. \

4.5 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Corrected PME description, added process unique value description
2. Added PME, DAT and process unique value to Process definition
3. Added PME bit as per ECO #43
4. Corrected DAT bit description as per ECO #40
5. Added DAT bit and FEN bit description
6. Converted to SDML
7. Added ECO #18, #21
8. Changed 'CC' to 'PCC' in HWPCB
9. Integrate references for Console ECO #15

Revision 4.0, March 29, 1991

1. Remove references to ASTs as 'interrupts', substituting 'exception' where appropriate

Revision 3.0, Mar 2, 1990

1. Lower number of PAL scratch words from 23 to 7
2. Make ASN field be ignored on systems that do not implement ASNs
3. Change ASTRR to ASTSR
4. Change alignment of HWPCB

Revision 2.0, October 4, 1989

1. Add FEN, CC, and PAL scratch areas to HWPCB

Revision 1.0, May 23, 1989

1. First review distribution.

\
15comment>(edited 11-may-92)

Chapter 5

OpenVMS Internal Processor Registers, (II)

5.1 Internal Processor Registers

This chapter describes the OpenVMS Alpha Internal Processor Registers (IPRs). These registers are read and written with Move From Processor Register (MFPR) and Move To Processor Register (MTPR) instructions; see Section 2.6.

These instructions accept an input operand in R16 and return a result, if any, in R0. Registers R1, R16, and R17 are UNPREDICTABLE after a CALL_PAL MxPR routines. If a CALL_PAL MxPR routine does not return a result in R0, then R0 is also UNPREDICTABLE on return.

Some IPRs (for example, ASTSR, ASTEN, IPL) may be both read and written in a combined operation by performing an MTPR instruction.

Internal Processor Registers may or may not be implemented as actual hardware registers. An implementation may choose any combination of PALcode and hardware to produce the architecturally specified function.

Internal Processor Registers are only accessible from Kernel mode.

5.2 Stack Pointer Internal Processor Registers

The stack pointers for User, Supervisor, and Executive stacks are accessible as IPRs through the CALL_PAL MTPR and MFPR instructions. An implementation may retain some or all of these stack pointers only in the HWPCB. In this case, MTPR and MFPR for these registers must access the corresponding PCB locations. However, implementations that have these stack pointers in internal hardware registers are not required to access the corresponding HWPCB locations for MTPR and MFPR. The HWPCB locations get updated when a SWPCTX instruction is executed.

An implementation may also choose to keep the Kernel Stack Pointer (KSP) in an internal hardware register (labelled IPR_KSP); however, this register is not directly accessible through MTPR and MFPR instructions. Because access to the KSP requires Kernel mode, the actual KSP is the current mode stack pointer (R30); thus access to KSP is provided through R30 and no MTPR or MFPR access is required. PALcode routines can directly access IPR_KSP as needed.

At System Initialization, the value of the KSP is taken from the initial HWPCB (see Chapter 4).

5.3 IPR Summary

Table 5–1: Internal Processor Register (IPR) Summary

Register Name	Mnemonic	Access[1]	Input R16	Output R0	Context Switched
Address Space Number	ASN	R	—	number	yes
AST Enable	ASTEN	R/W*	mask	mask	yes
AST Summary Register	ASTSR	R/W*	mask	mask	yes
Data Align Trap Fixup	DATFX	W	value	—	yes
Floating-point Enable	FEN	R/W	value	value	yes
Interprocessor Int. Request	IPIR	W	number	—	no
Interrupt Priority Level	IPL	R/W*	value	value	no
Machine Check Error Summary	MCES	R/W	value	value	no
Performance Monitor	PERFMON	W*	IMP	IMP	no
Privileged Context Block Base	PCBB	R	—	address	no
Processor Base Register	PRBR	R/W	value	value	no
Page Table Base Register	PTBR	R	—	frame	yes
System Control Block Base	SCBB	R/W	frame	frame	no
Software Int. Request Register	SIRR	W	level	—	no
Software Int. Summary Register	SISR	R	—	mask	no
TB Check	TBCHK	R	number	status	no
TB Invalid. All	TBIA	W	—	—	no
TB Invalid. All Process	TBIAP	W	—	—	no
TB Invalid. Single	TBIS	W	address	—	no
TB Invalid. Single Data	TBISD	W	address	—	no
TB Invalid. Single Instruct.	TBISI	W	address	—	no
Kernel Stack Pointer	KSP	None	—	—	yes
Exec Stack Pointer	ESP	R/W	address	address	yes
Supervisor Stack Pointer	SSP	R/W	address	address	yes
User Stack Pointer	USP	R/W	address	address	yes
Virtual Page Table Base	VPTB	R/W	address	address	no
Who-Am-I	WHAMI	R	—	number	no

[1]Access symbols are defined in Table 5–2

Table 5–2: Internal Processor Register (IPR) Access Summary

Access Type	Meaning
R	Access by MFPR only.
W	Access by MTPR only.
R/W	Access by MFPR or MTPR.
W*	Read and Write access accomplished by MTPR; see Section 5.1 for details.
R/W*	Access by MFPR or MTPR. Read and Write access accomplished by MTPR; see Section 5.1 for details.
None	Not accessible by MTPR or MFPR; accessed by PALcode routines as needed.

5.3.1 Address Space Number (ASN)

Access:

Read

Operation:

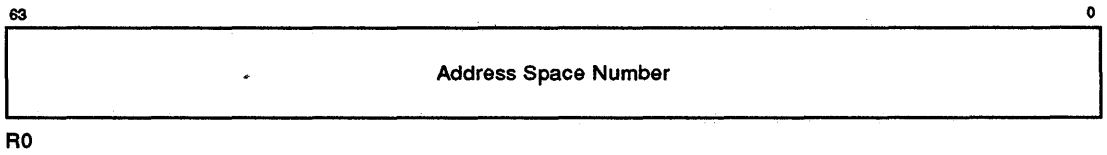
```
IF {ASN are implemented} THEN
   R0 ←  ZEXT(ASN)
ELSE
   R0 ←  0
```

Value at System Initialization:

Zero

Format:

Figure 5–1: Address Space Number Register (ASN)



Description:

Address Space Numbers (ASNs) are used to further qualify Translation Buffer references; see Chapter 3. If ASNs are implemented, the current ASN may be read by executing an MFPR instruction specifying ASN.

As processes are scheduled for execution, the ASN for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction; see Chapters 2 and 4.

The ASN register is an implicit operand to the CALL_PAL MFPR_IPR, TBCHK, and TBISx PALcode instructions, in which it is used to qualify the virtual address supplied in R16.

5.3.2 AST Enable (ASTEN)

Access:

Read

Write*

Operation:

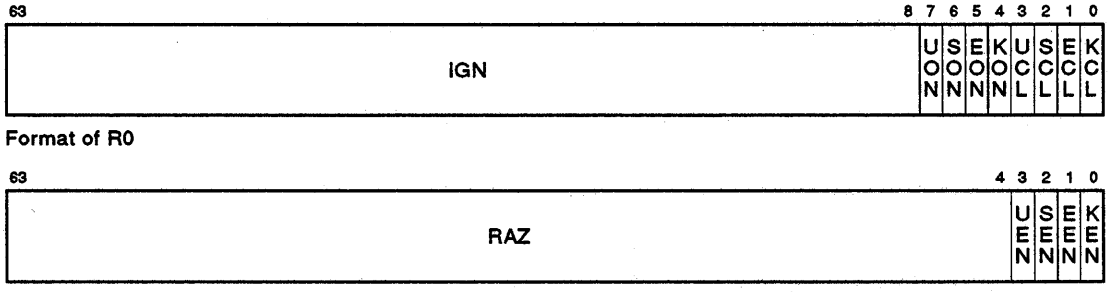
```
R0 ←  ZEXT (ASTEN<3:0>)              ! Read (MFPR)
R0 ←  ZEXT(ASTEN<3:0>)               ! Write* (MTPR)
ASTEN<3:0> ←  {{ASTEN<3:0> AND R16<3:0>} OR R16<7:4>}
{check for pending ASTs}
```

Value at System Initialization:

Zero

Format:

Figure 5–2: AST Enable Register (ASTEN)



Description:

The AST Enable Register records the AST enable state for each of the modes: Kernel (KEN), Executive (EEN), Supervisor (SEN) and User (UEN). By writing R16 appropriately and then executing an MTPR instruction specifying ASTEN, the value of ASTEN may be simultaneously read and modified. R16 contains bit masks used to determine the new value of ASTEN:

- Bits R16<0> and R16<4> control the new state of Kernel enable.
- Bits R16<1> and R16<5> control the new state of Executive enable.

- Bits R16<2> and R16<6> control the new state of Supervisor enable.
- Bits R16<3> and R16<7> control the new state of User enable.

An MFPR to ASTEN reads the current value of the ASTEN and returns this value in R0.

An MTPR to ASTEN begins by reading the current value of ASTEN and returning this value in R0. The current value of ASTEN is then ANDed with bits R16<3:0>; these bits preserve (if set to '1') or clear (if equal to '0') the current state of their corresponding enable modes. The value produced by this operation is then ORed with bits R16<7:4>; these bits turn on (if set to '1') or do not affect (if equal to '0') their corresponding enable modes. The resulting value is then written to the ASTEN.

NOTE

All AST enables can be cleared by loading a zero into R16 and executing an MTPR instruction specifying ASTEN. To enable an AST for a given mode, load R16 with a mask that has bits <3:0> set and one of the bits <7:4> corresponding to the AST mode to be set. Then execute an MTPR instruction specifying ASTEN.

\ ASTEN is not present in the VAX architecture. It was added to the Alpha architecture to allow software (especially nonprivileged software) to enable and disable ASTs efficiently for the current mode via the SWASTEN instruction. It is anticipated that, with multitasking, it will become extremely important to be able to enable and disable ASTs in an efficient manner in shareable runtime support routines.\

As processes are scheduled for execution, the state of the AST enables for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction. The Swap AST Enable (SWASTEN) instruction can be used to change the enable state for the current access mode; See Chapters 2 and 4.

5.3.3 AST Summary Register (ASTSR)

Access:

Read

Write*

Operation:

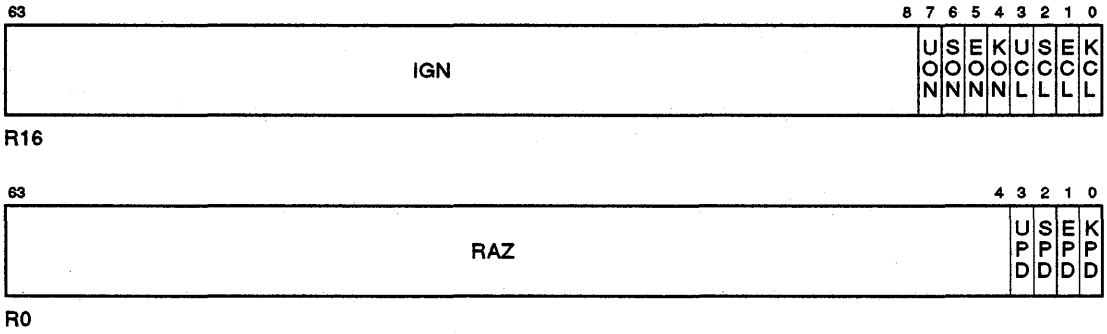
```
R0 ←  ZEXT(ASTSR<3:0>)       ! Read (MFPR)
R0 ←  ZEXT(ASTSR<3:0>)       ! Write* (MTPR)
ASTSR<3:0> ←  {{ASTSR<3:0> AND R16<3:0>} OR R16<7:4>}
{check for pending ASTs}
```

Value at System Initialization:

Zero

Format:

Figure 5–3: AST Summary Register (ASTSR)



Description:

The AST Summary Register records the AST pending state for each of the modes: Kernel (KPD), Executive (EPD), Supervisor (SPD), and User (UPD).

By writing R16 appropriately and then executing an MTPR instruction specifying ASTSR, the value of ASTSR may be simultaneously read and modified. R16 contains bit masks used to determine the new value of ASTSR:

- Bits R16<0> and R16<4> control the new state of Kernel pending.

- Bits R16<1> and R16<5> control the new state of Executive pending.
- Bits R16<2> and R16<6> control the new state of Supervisor pending.
- Bits R16<3> and R16<7> control the new state of User pending.

An MFPR reads the current value of ASTSR and returns this value in R0.

An MTPR to ASTSR begins by reading the current value of ASTSR and returning this value in R0. The current value of ASTSR is then ANDed with bits R16<3:0>; these bits preserve (if set to '1') or clear (if equal to '0') the current state of their corresponding pending modes. The value produced by this operation is then ORed with bits R16<7:4>; these bits turn on (if set to '1') or do not affect (if equal to '0') their corresponding pending modes. The resulting value is then written to the ASTSR.

NOTE

All AST requests can be cleared by loading a zero in R16 and executing an MTPR instruction specifying ASTSR. To request an AST for a given mode, load R16 with a mask that has bits <3:0> set and one of the bits <7:4> corresponding to the AST mode to be set. Then execute an MTPR instruction specifying ASTSR.

As processes are scheduled for execution, the pending AST state for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction; see Chapters 2 and 4.

When the processor IPL is less than 2, and proper enabling conditions are present, an AST interrupt is initiated at IPL 2 and the corresponding access mode bit in ASTSR is cleared; see Section 6.7.6.

5.3.4 Data Alignment Trap Fixup (DATFX)

Access:

Write

Operation:

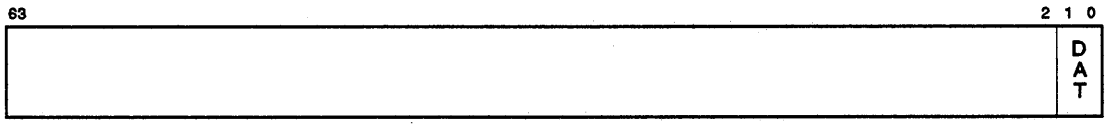
```
DATFX ← R16<0>
(HWPCB+56)<63> ← DATFX
```

Value at System Initialization:

Zero

Format:

Figure 5–4: Data Alignment Trap Fixup (DATFX)



Description:

Data Alignment traps are fixed up in PALcode and are reported to the operating system under the control of the DAT bit. If the bit is zero, the trap is reported. For the LDx_L and STx_C instructions, no fixup is possible and an illegal operand exception is generated. For the description of the data alignment traps, see Section 6.6.

5.3.5 Floating Enable (FEN)

Access:

Read/Write

Operation:

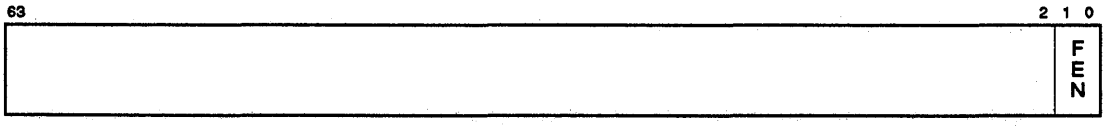
```
R0 ←  ZEXT(FEN)                 ! Read
FEN ←  R16<0>                   ! Write
(HWPCB+56)<0> ←  FEN            ! Update PCB on Write
```

Value at System Initialization:

Zero

Format:

Figure 5–5: Floating Enable (FEN) Register



Description:

The Floating-point unit can be disabled. If the Floating Enable Register (FEN) is zero, all instructions that have floating registers as operands cause a Floating-point disabled fault; see Section 6.3.1.1.

5.3.6 Interprocessor Interrupt Request (IPIR)

Access:

Write

Operation:

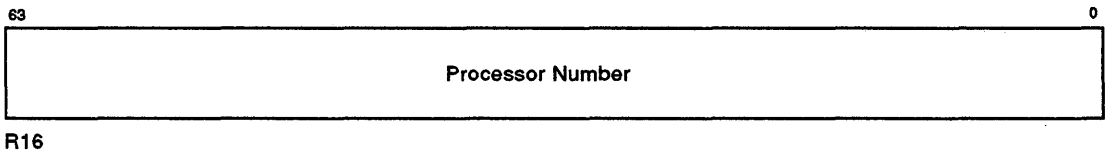
```
IPIR ← R16
```

Value at System Initialization:

Not applicable

Format:

Figure 5–6: Interprocessor Interrupt Request Register (IPIR)



Description:

An interprocessor interrupt can be requested on a specified processor by writing that processor's number into the IPIR register through an MTPR instruction. The interrupt request is recorded on the target processor and is initiated when proper enabling conditions are present.

PROGRAMMING NOTE

The interrupt need not be initiated before the next instruction is executed on the requesting processor, even if the requesting processor is also the target processor for the request.

For additional information on interprocessor interrupts, see Section 6.4.5.1.

5.3.7 Interrupt Priority Level (IPL)

Access:

Read/Write*

Operation:

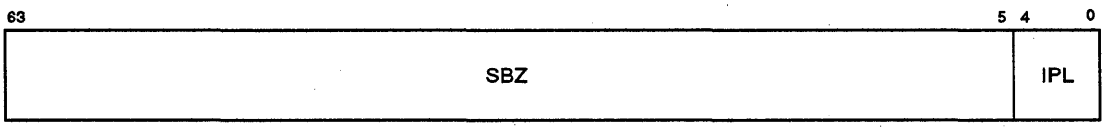
```
R0 ←  ZEXT(PS<IPL>)      ! Read
R0 ←  ZEXT(PS<IPL>)      ! Write*
PS<IPL> ←  R16<4:0>      ! Write
{check for pending ASTs or interrupts}
```

Value at System Initialization:

31

Format:

Figure 5–7: Interrupt Priority Level (IPL)



Description:

An MFPR IPL returns the current interrupt priority level in R0. An MTPR IPL returns the current interrupt priority level in R0 and sets the interrupt priority level to the value in R16. If proper enabling conditions are present, an interrupt or AST is initiated prior to issuing the next instruction; see Sections 6.4.1 and 6.7.6. R16<63:5> are defined as RAZ/SBZ. Therefore, the presence of non-zero bits upon write in R16<63:5> may cause UNDEFINED results.

5.3.8 Machine Check Error Summary Register (MCES)

Access:

Read/Write

Operation:

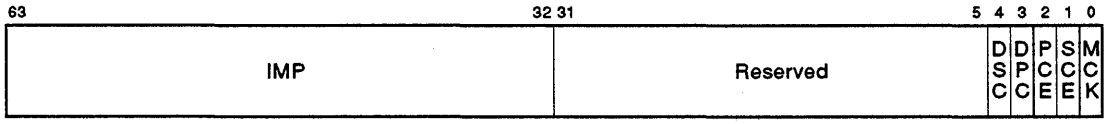
```
R0 ←  ZEXT(MCES)                                  ! Read

IF {R16<0> EQ 1} THEN MCES<0> ←  0   ! Write
IF {R16<1> EQ 1} THEN MCES<1> ←  0
IF {R16<2> EQ 1} THEN MCES<2> ←  0
MCES<3> ←  R16<3>
MCES<4> ←  R16<4>
```

Value at System Initialization:

Zero

Format:

Figure 5–8: Machine Check Error Summary Register (MCES)



Description:

The use of the MCES IPR is described in Section 6.5.

MCES<0> is set by the hardware or PALcode when a processor or system machine check occurs. MCES<1> is set by the hardware or PALcode when a system correctable error occurs. MCES<2> is set by the hardware or PALcode when a processor correctable error occurs. Writing a 1 to any of these three bits clears that bit.

MCES<0> is cleared by the operating system machine check error handler and used by the hardware or PALcode to detect double machine checks. MCES<1> and MCES<2> are cleared by the operating system system or processor system correctable error handlers; these bits are used to indicate that the associated correctable error logout area may be reused by hardware or PALcode. In the event

of double correctable errors, PALcode does not overwrite the logout area and does not force the processor to enter console I/O mode; see Section 6.5.1.

MCES<4:3> are used to disable reporting of correctable errors. When set, the error is corrected, but no system correctable error interrupt or processor correctable machine check is generated.

Implementation dependent (IMP) bits may be used to report implementation specific errors.

5.3.9 Performance Monitoring Register (PERFMON)

Access:

Write*

Operation:

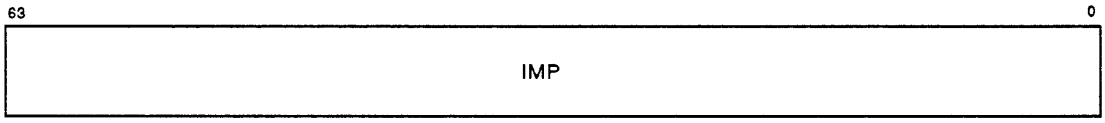
```
! R<16> contains implementation specific input values
! R<0>  may return implementation specific values
! Operations and actions taken are implementation specific
```

Value at System Initialization:

Implementation Dependent

Format:

Figure 5–9: Performance Monitoring Register (PERFMON)



Description:

The arguments and actions of this performance monitoring function are platform and chip dependent. The functions, when defined for an implementation, are to be registered in *Appendix E*.

R<16> contains implementation dependent input values. Implementation specific values may be returned in R<0>.

5.3.10 Privileged Context Block Base (PCBB)

Access:

Read

Operation:

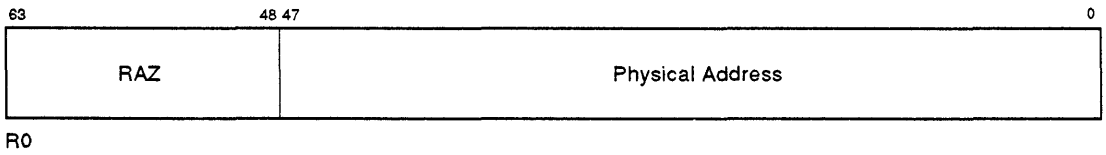
```
R0 ← ZEXT(PCBB)
```

Value at System Initialization:

Address of processor's bootstrap HWPCB

Format:

Figure 5–10: Privileged Context Block Base Register (PCBB)



Description:

The Privileged Context Block Base Register contains the physical address of the privileged context block for the current process. It may be read by executing an MFPR instruction specifying PCBB.

PCBB is written by the Swap Privileged Context (SWPCTX) instruction; see Chapters 2 and 4.

5.3.11 Processor Base Register (PRBR)

Access:

Read/Write

Operation:

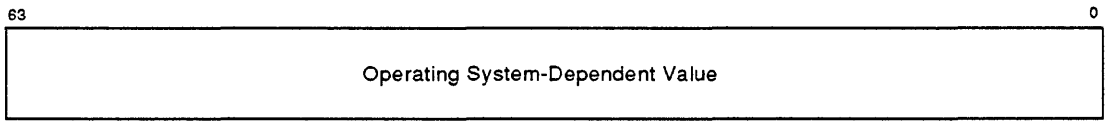
```
R0 ← PRBR                    ! Read
PRBR ← R16                   ! Write
```

Value at System Initialization:

UNPREDICTABLE

Format:

Figure 5–11: Processor Base Register (PRBR)



Description:

In a multiprocessor system, it is desirable for the operating system to be able to locate a processor-specific data structure in a simple and straightforward manner. The Processor Base Register provides a quadword of operating system-dependent state that can be read and written via MFPR and MTPR instructions that specify PRBR.

5.3.12 Page Table Base Register (PTBR)

Access:

Read

Operation:

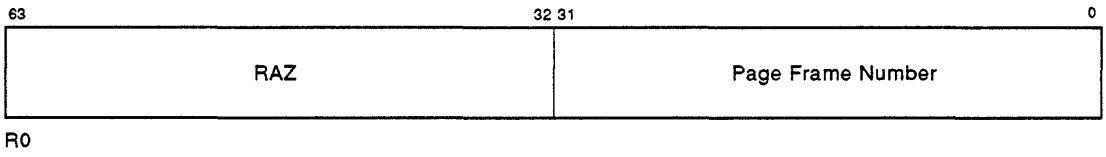
```
R0 ← PTBR
```

Value at System Initialization:

Value in the bootstrap HWPCB

Format:

Figure 5–12: Page Table Base Register (PTBR)



Description:

The Page Table Base Register contains the page frame number of the first-level page table for the current process. It may be read by executing an MFPR instruction specifying PTBR; see Chapter 3.

As processes are scheduled for execution, the PTBR for the next process to execute is loaded using the Swap Privileged Context (SWPCTX) instruction; see Chapters 2 and 4.

5.3.13 System Control Block Base (SCBB)

Access:

Read/Write

Operation:

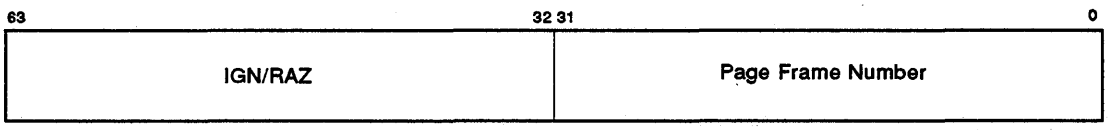
```
R0 ←  ZEXT(SCBB)          ! Read
SCBB ←  R16               ! Write
```

Value at System Initialization:

UNPREDICTABLE

Format:

Figure 5–13: System Control Block Base Register (SCBB)



Description:

The System Control Block Base Register holds the Page Frame Number (PFN) of the System Control Block, which is used to dispatch exceptions and interrupts, and may be read and written by executing MFPR and MTPR instructions that specify SCBB; see Section 6.6.

When SCBB is written, the specified physical address must be the PFN of a page which is neither in I/O space nor non-existent memory, or UNDEFINED operation will result.

5.3.14 Software Interrupt Request Register (SIRR)

Access:

Write

Operation:

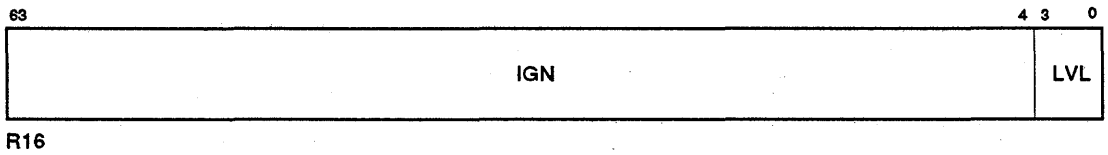
```
IF R16<3:0> NE 0 THEN
    SISR<R16<3:0>> ← 1
```

Value at System Initialization:

Not applicable

Format:

Figure 5–14: Software Interrupt Request Register (SIRR)



Description:

A software interrupt may be requested for a particular Interrupt Priority Level (IPL) by executing an MTPR instruction specifying SIRR. Software interrupts may be requested at levels 0 through 15 (requests at level 0 are ignored).

An MTPR SIRR sets the bit corresponding to the specified interrupt level in the Software Interrupt Summary Register (SISR).

If proper enabling conditions are present, a software interrupt is initiated prior to issuing the next instruction; see Sections 6.4.1 and 6.7.6.

5.3.15 Software Interrupt Summary Register (SISR)

Access:

Read

Operation:

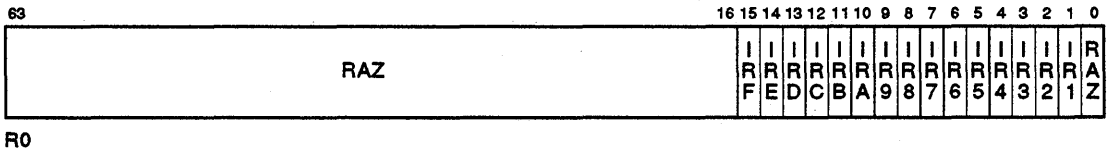
```
R0 ← ZEXT(SISR<15:0>)
```

Value at System Initialization:

Zero

Format:

Figure 5–15: Software Interrupt Summary Register (SISR)



Description:

The Software Interrupt Summary Register records the interrupt pending state for each of the interrupt levels 1 through 15. The current interrupt pending state may be read by executing an MFPR instruction specifying SISR.

MTPR SIRR (see SIRR) requests an interrupt at a particular interrupt level and sets the corresponding pending bit in SISR.

When the processor IPL falls below the level of a pending request, an interrupt is initiated and the corresponding bit in SISR is cleared; see Sections 6.4.1 and 6.7.6.

5.3.16 Translation Buffer Check (TBCHK)

Access:

Read

Operation:

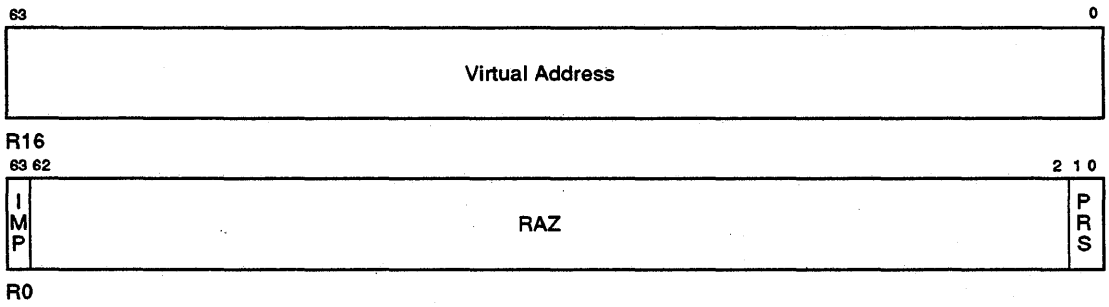
```
R0 ← 0
IF {implemented} THEN
    R0<0> ← {entry in TB for VA in R16}
ELSE
    R0<63> ← 1
```

Value at System Initialization:

Correct results are always returned

Format:

Figure 5–16: Translation Buffer Check Register (TBCHK)



Description:

The Translation Buffer Check Register provides the capability to determine if a virtual address is present in the Translation Buffer by executing an MFPR instruction specifying TBCHK; see Chapter 3.

The virtual address to be checked is specified in R16 and may be any address within the desired page. If ASNs are implemented, only those Translation Buffer entries which are associated with the current value of the ASN IPR will be checked for the virtual address. The value read contains an indication of whether the function is implemented and whether the virtual address is present in the Translation Buffer.

If the function is not implemented, a value is returned with bit <63> set and bit <0> clear. Otherwise, a value is returned with bit <63> clear, and with bit <0> indicating whether the virtual address is present in (1) or absent from (0) the Translation Buffer.

The TBCHK Register can be used by system software for working set management.

5.3.17 Translation Buffer Invalidate All (TBIA)

Access:

Write

Operation:

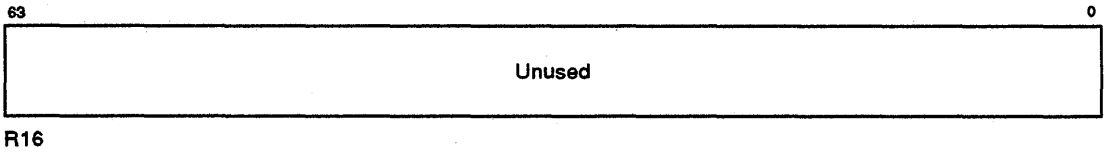
```
{Invalidate all TB entries}
```

Value at System Initialization:

Not applicable

Format:

Figure 5–17: Translation Buffer Invalidate All Register (TBIA)



Description:

The Translation Buffer Invalidate All Register provides the capability to invalidate all entries in the Translation Buffer by executing an MTPR instruction specifying TBIA; see Chapter 3.

5.3.18 Translation Buffer Invalidate All Process (TBIAP)

Access:

Write

Operation:

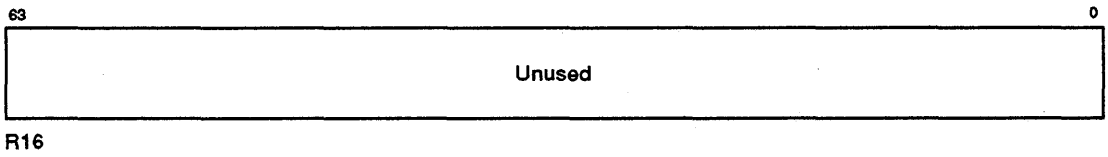
```
{Invalidate all TB entries with PTE<ASM> clear}
```

Value at System Initialization:

Not applicable

Format:

Figure 5–18: Translation Buffer Invalidate All Process Register (TBIAP)



Description:

The Translation Buffer Invalidate All Process Register provides the capability to invalidate all entries in the Translation Buffer that do not have the ASM bit set by executing an MTPR instruction specifying TBIAP; see Chapter 3.

Notes:
More entries may be invalidated by this operation. For example some implementations may flush the entire TB on a TBIAP.

5.3.19 Translation Buffer Invalidate Single (TBISx)

Access:

Write

Operation:

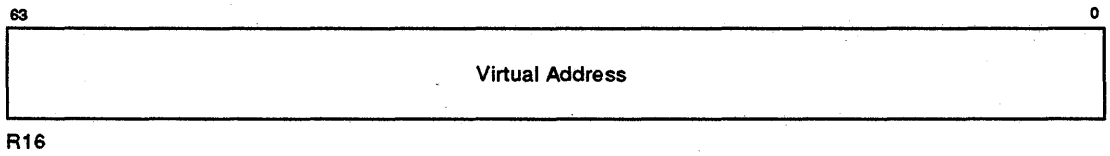
```
TBIS:
    {Invalidate single Data TB entry using R16}
    {Invalidate single Instruction TB entry using R16}
TBISD:
    {Invalidate single Data TB entry using R16}
TBISI:
    {Invalidate single Instruction TB entry using R16}
```

Value at System Initialization:

Not applicable

Format:

Figure 5–19: Translation Buffer Invalidate Single (TBIS)



Description:

The Translation Buffer Invalidate Single Registers provide the capability to invalidate a single entry in the Instruction Translation Buffer (TBISI), the Data Translation Buffer (TBISD), or both translation buffers (TBIS). The virtual address to be invalidated is passed in R16 and may be any address within the desired page.

Notes:
More than the single entry may be invalidated by this operation. For example some implementations may flush the entire TB on a TBIS. As a result, if the specified address does not match any entry in the Translation Buffer, then it is implementation-dependent whether the state of the Translation Buffer is affected by the operation.

5.3.20 Executive Stack Pointer (ESP)

Access:

Read/Write

Operation:

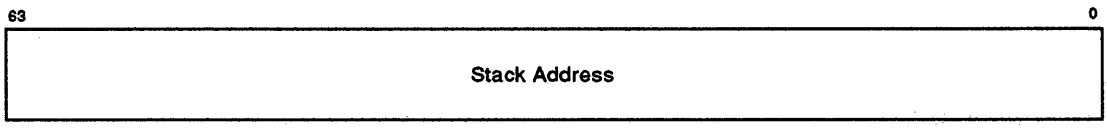
```
IF {internal registers for stack pointers}  THEN      ! Read
    R0 ←  ESP
ELSE
    R0 ←  (IPR_PCBB + HWPCB_ESP)
IF {internal registers for stack pointers}  THEN      ! Write
    ESP ←  R16
ELSE
    (IPR_PCBB + HWPCB_ESP) ←  R16
```

Value at System Initialization:

Value in the initial HWPCB

Format:

Figure 5–20: Executive Stack Pointer (ESP)



Description:

This register allows the stack pointer for Executive mode (ESP) to be read and written via MFPR and MTPR instructions that specify ESP.

The current stack pointer may be read and written directly by specifying scalar register SP (R30).

As processes are scheduled for execution, the stack pointers for the next process to execute are loaded using the Swap Privileged Context (SWPCTX) instruction; see Section 2.6 and Chapter 4.

5.3.21 Supervisor Stack Pointer (SSP)

Access:

Read/Write

Operation:

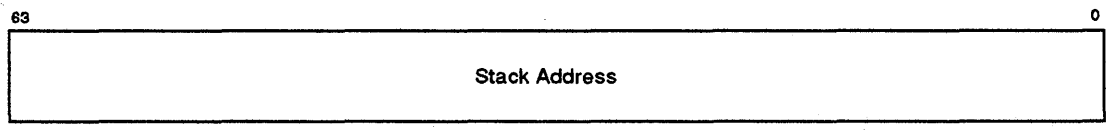
```
IF {internal registers for stack pointers}  THEN      ! Read
    R0 ←  SSP
ELSE
    R0 ←  (IPR_PCBB + HWPCB_SSP)
IF {internal registers for stack pointers}  THEN      ! Write
    SSP ←  R16
ELSE
    (IPR_PCBB + HWPCB_SSP) ←  R16
```

Value at System Initialization:

Value in the initial HWPCB

Format:

Figure 5–21: Supervisor Stack Pointer (SSP)



Description:

This register allows the stack pointer for Supervisor mode (SSP) to be read and written via MFPR and MTPR instructions that specify SSP.

The current stack pointer may be read and written directly by specifying scalar register SP (R30).

As processes are scheduled for execution, the stack pointers for the next process to execute are loaded using the Swap Privileged Context (SWPCTX) instruction; see Section 2.6 and Chapter 4.

5.3.22 User Stack Pointer (USP)

Access:

Read/Write

Operation:

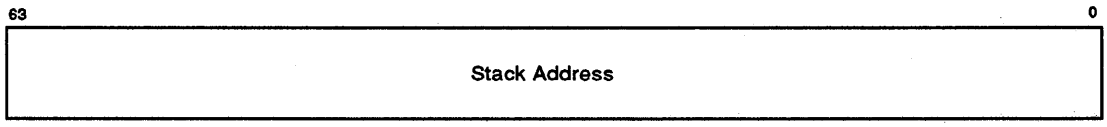
```
IF {internal registers for stack pointers}  THEN      ! Read
    R0 ←  USP
ELSE
    R0 ←  (IPR_PCBB + HWPCB_USP)
IF {internal registers for stack pointers}  THEN      ! Write
    USP ←  R16
ELSE
    (IPR_PCBB + HWPCB_USP) ←  R16
```

Value at System Initialization:

Value in the initial HWPCB

Format:

Figure 5–22: User Stack Pointer (USP)



Description:

This register allows the stack pointer for User mode (USP) to be read and written via MFPR and MTPR instructions that specify USP.

The current stack pointer may be read and written directly by specifying scalar register SP (R30).

As processes are scheduled for execution, the two stack pointers for the next process to execute are loaded using the Swap Privileged Context (SWPCTX) instruction; see Section 2.6 and Chapter 4.

5.3.23 Virtual Page Table Base (VPTB)

Access:

Read/Write

Operation:

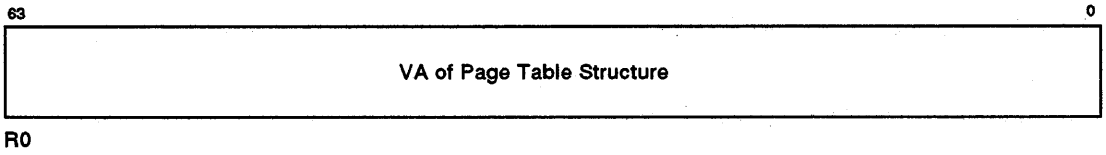
```
R0 ← VPTB                    ! Read
VPTB ← R16                   ! Write
```

Value at System Initialization:

Initialized by the console in the bootstrap address space.

Format:

Figure 5–23: Virtual Page Table Base Register (VPTB)



Description:

The Virtual Page Table Base Register contains the virtual address of the base of the entire three-level Page table structure. It may be read by executing an MFPR instruction specifying VPTB. It is written at system initialization using an MTPR instruction specifying VPTB. See Section 3.7.2 \ and *Platform Section, Chapter 4* \ for initialization considerations.

5.3.24 Who-Am-I (WHAMI)

Access:

Read

Operation:

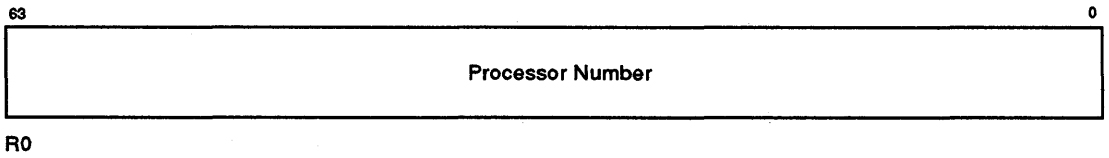
```
R0 ← WHAMI
```

Value at System Initialization:

Processor number

Format:

Figure 5–24: Who-Am-I Register (WHAMI)



Description:

The Who-Am-I Register provides the capability to read the current processor number by executing an MFPR instruction specifying WHAMI. The processor number returned is in the range 0 to the number of processors minus one that can be configured in the system. Processor number FFFF FFFF FFFF $FFFF_{16}$ is reserved.

The current processor number is useful in a multiprocessing system to index arrays that store per processor information. Such information is operating system dependent.

5.4 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Added changes to MCES for ECO #45
2. Added Perfmon ipr description and entry in summary table
3. Added DATFX related ecos #30, #40
4. Added bit field to FEN reference to PCBB as a result of datfx ecos
5. Added VPTB register
6. Rewrite of MCES description
7. Converted to SDML
8. Added ECO #16, #17, #20, #23, #24
9. Integrate references for Console ECO #15

Revision 4.0, March 29, 1991

1. MTPR IPL returns old IPL in R0
2. Typos
3. Change MCES IGN/RAZ field to IMP
4. Describe how to clear and set mode enable bits with MTPR
5. Change text for ASTSR description to indicate future action for mode set
6. Change ASTEN and ASTSR to access type Read/Write
7. Modify (subtly) note under IPIR to avoid confusion about timing relation between processors
8. Clarify what value to load into IPIR to select a particular target
9. Change 'Value at System Initialization' from 'UNDEFINED' to 'UNPREDICTABLE' for PRBR and SCBB
10. Note effect of writing TBIS with an address that does not match any TB entry
11. Note that ASN is an implicit operand to a MFPR TBCHK instruction
12. Emphasise distinction between SIRR and SISR
13. Reworked IPR table to show which IPRs are context-switched and which are not.
14. Remove references to ASTs as 'interrupts', substituting 'exception' where appropriate
15. Insert spaces into long hex and binary values to improve legibility
16. Clarify obscure use of MTPR to both read and write certain IPRs
17. Illustrate R16 bits used to 'gate' ASTEN and ASTSR contents into R0

18. Add pointer in the IPIR IPR section pointing to Interprocessor Interrupt material in Chapter 6
19. Add Kernel Stack Pointer as an internal processor register
20. Modify definition of Absolute Time register and BB_WATCH entity.
21. Changed IPR Summary Table and added R/W* description
22. Specified all systems that support VAX or ULTRIX must have a BB_WATCH
23. Clarified value written to IPIR to select a processor

Revision 3.0, March 2, 1990

1. Remove ASTRR and make ASTEN/ASTSR read/write
2. Add TBIAP
3. Remove ASN from TBIx and TBCHK
4. Remove R17 as input to MxPR's
5. Reserve processor number FFFF FFFF FFFF $FFFF_{16}$

Revision 2.0, October 4, 1989

1. Remove ICIE, IPIE, ISP, KSP, SID, SSN, and TOY
2. Add AT and FEN
3. Change range of WHAMI
4. Remove stack alignment comments
5. Change registers used to match calling standard

Revision 1.0, March 15, 1989

1. First review distribution.

Chapter 6

OpenVMS Exceptions, Interrupts, and Machine Checks (II)

6.1 Introduction

At certain times during the operation of a system, events within the system require the execution of software outside the explicit flow of control. When such an exceptional event occurs, an Alpha processor forces a change in control flow from that indicated by the current instruction stream. The notification process for such events is of one of three types:

- Exceptions

  These events are relevant primarily to the currently executing process and normally invoke software in the context of the current process. The three types of exceptions are faults, arithmetic traps, and synchronous traps. Exceptions are described in Section 6.3.

- Interrupts

  These events are primarily relevant to other processes, or to the system as a whole, and are typically serviced in a system-wide context.

  Some interrupts are of such urgency that they require high-priority service, while others must be synchronized with independent events. To meet these needs, each processor has priority logic that grants interrupt service to the highest priority event at any point in time. Interrupts are described in Section 6.4.

- Machine Checks

  These events are generally the result of serious hardware failure. The registers and memory are potentially in an indeterminate state such that the instruction execution cannot necessarily be correctly restarted, completed, simulated, or undone. Machine checks are described in Section 6.5.

For all such events, the change in flow of control involves changing the Program Counter (PC), possibly changing the execution mode (current mode) and/or interrupt priority level (IPL) in the Processor Status (PS), and saving the old values of the PC and PS. The old values are saved on the target stack as part of an Exception, Interrupt, or Machine Check Stack Frame. Collectively, those elements are described in Section 6.2.

The service routines that handle exceptions, interrupts, and machine checks are specified by entry points in the System Control Block (SCB), described in Section 6.6.

Return from an exception, interrupt, or machine check, is done via the CALL_PAL REI instruction. As part of its work, CALL_PAL REI restores the saved values of PC and PS and pops them off the stack.

6.1.1 Contrast Between Exceptions, Interrupts, and Machine Checks

Generally, exceptions, interrupts, and machine checks are similar. However, there are four important differences:

1. An exception condition is caused by the execution of an instruction. An interrupt is caused by some activity in the system that may be independent of any instruction. A machine check is associated with a hardware error condition.
2. The IPL of the processor is not changed when the processor initiates an exception. The IPL is always raised when an interrupt is initiated. The IPL is always raised when a machine check is initiated, and for all machine checks other than system correctable, is raised to 31 (highest priority level). (For system correctable machine checks, the IPL is raised to 20.)
3. Exceptions are always initiated immediately, no matter what the processor IPL is. Interrupts are deferred until the processor IPL drops below the IPL of the requesting source. Machine checks can be initiated immediately or deferred, depending on error conditions.
4. Some exceptions can be selectively disabled by selecting instructions that do not check for exception conditions. If an exception condition occurs in such an instruction, the condition is totally ignored and no state is saved to signal that condition at a later time.

   If an interrupt request occurs while the processor IPL is equal to or greater than that of the interrupting source, the condition will eventually initiate an interrupt if the interrupt request is still present and the processor IPL is lowered below that of the interrupting source.

   Machine checks cannot be disabled. Machine checks can be initiated immediately or deferred, depending on the error condition. Also, they can be deliberately generated by software.

6.1.2 Exceptions, Interrupts, and Machine Checks Summary

The table below summarizes the actions taken on an exception, interrupt, or machine check. The remaining sections in this chapter describe these in greater detail.

- The "SavedPC" column describes what is saved in the "PC" field of the exception or interrupt or machine check stack frame. Here,
  1. "Current" indicates the PC of the instruction at which the exception or interrupt or machine check was taken, while
  2. "Next" indicates the PC of the successor instruction.
- The "NewMode" column specifies the mode and stack that the exception or interrupt or machine check routine will start with. For change mode traps, "MostPrv" indicates the more privileged of the current and new modes.

- The "R2" column specifies the value with which R2 is loaded, after its original value has been saved in the exception or interrupt or machine check stack frame. The SCB vector quadword, "SCBv", is loaded into R2 for all interrupts and exceptions and machine checks.
- The "R3" column specifies the value with which R3 is loaded, after its original value has been saved in the exception or interrupt or machine check stack frame. The SCB parameter quadword, "SCBp", is loaded into R3 for all interrupts and exceptions and machine checks.
- The "R4" column specifies the value with which R4 is loaded, after its original value has been saved in the exception or interrupt or machine check stack frame. If the "R4" column is blank the value in R4 is UNPREDICTABLE on entry to an interrupt or exception. Here,
  1. "VA" indicates the exact virtual address which triggered a memory management fault or data alignment trap.
  2. "Mask" indicates the Register Write Mask.
  3. "LAOff" indicates the offset from the base of the logout area in the HWRPB; see Section 6.5.2.
- The "R5" column specifies the value with which R5 is loaded, after its original value has been saved in the exception or interrupt or machine check stack frame. If the "R5" column is blank the value in R5 is UNPREDICTABLE on entry to an interrupt or exception or machine check. Here,
  1. "MMF" indicates the Memory Management Flags.
  2. "Exc" indicates the Exception Summary parameter.
  3. "RW" indicates Read/Load =0 Write/Store =1 for data align traps

Table 6–1: Exceptions, Interrupts, and Machine Checks Summary

	SavedPC	NewMode	R2	R3	R4	R5
Exceptions - Faults						
Floating Disabled Fault	Current	Kernel	SCBv	SCBp		
Memory Management Faults						
Access Control Violation	Current	Kernel	SCBv	SCBp	VA	MMF
Translation Not Valid	Current	Kernel	SCBv	SCBp	VA	MMF
Fault on Read	Current	Kernel	SCBv	SCBp	VA	MMF
Fault on Write	Current	Kernel	SCBv	SCBp	VA	MMF
Fault on Execute	Current	Kernel	SCBv	SCBp	VA	MMF

Table 6–1 (Cont.): Exceptions, Interrupts, and Machine Checks Summary

	SavedPC	NewMode	R2	R3	R4	R5
Exceptions - Arithmetic Traps						
Arithmetic Traps	Next	Kernel	SCBv	SCBp	Mask	Exc
Exceptions - Synchronous Traps						
Breakpoint Trap	Next	Kernel	SCBv	SCBp		
Bugcheck Trap	Next	Kernel	SCBv	SCBp		
Change Mode to K/E/S/U	Next	MostPrv	SCBv	SCBp		
Illegal Instruction	Next	Kernel	SCBv	SCBp		
Illegal Operand	Next	Kernel	SCBv	SCBp		
Data Alignment Trap	Next	Kernel	SCBv	SCBp	VA	RW
Interrupts						
Asynch System Trap (4)	Current	Kernel	SCBv	SCBp		
Interval Clock	Current	Kernel	SCBv	SCBp		
Interprocessor Interrupt	Current	Kernel	SCBv	SCBp		
Software Interrupts	Current	Kernel	SCBv	SCBp		
Performance monitor	Current	Kernel	SCBv	SCBp	IMP	IMP
Passive Release	Current	Kernel	SCBv	SCBp		
Powerfail	Current	Kernel	SCBv	SCBp		
I/O Device	Current	Kernel	SCBv	SCBp		
Machine Checks						
Processor Correctable	Current	Kernel	SCBv	SCBp	LAOff	
System Correctable	Current	Kernel	SCBv	SCBp	LAOff	
System	Current	Kernel	SCBv	SCBp	LAOff	
Processor	Current	Kernel	SCBv	SCBp	LAOff	

6.2 Processor State and Exception/Interrupt/Machine Check Stack Frame

Processor state consists of a quadword of privileged information called the Processor Status (PS) and a quadword containing the Program Counter (PC), which is the virtual address of the next instruction.

When an exception, interrupt, or machine check is initiated, the current processor state during the exception, interrupt, or machine check must be preserved. This is accomplished by automatically pushing the PS and the PC on the target stack.

Subsequently, instruction execution can be continued at the point of the exception, interrupt, or machine check by executing a CALL_PAL REI instruction; see Chapter 2.

Process context such as memory mapping information is not saved or restored on each exception, interrupt, or machine check. Instead, it is saved and restored when process context switching is performed. Other processor status is changed even less frequently; see Chapter 4.

6.2.1 Processor Status

The PS can be explicitly read with the CALL_PAL RD_PS instruction. The PS<SW> field can be explicitly written with the CALL_PAL WR_PS_SW instruction. See Section 2.1.

The terms current PS and saved PS are used to distinguish between this status information when it is stored internal to the processor and when copies of it are materialized in memory.

Figure 6–1: Current Processor Status (PS Register)

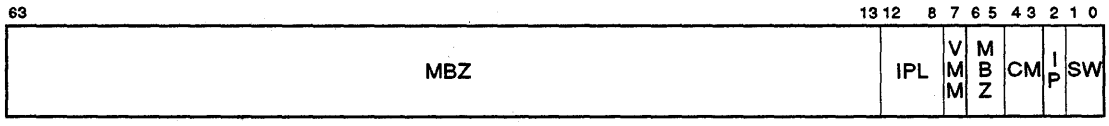


Figure 6–2: Saved Processor Status (PS on Stack)

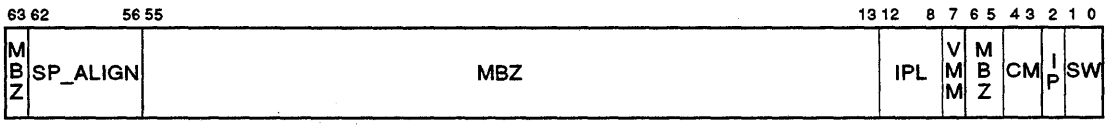


Table 6–2: Processor Status Register Summary

Bits	Description
1:0	Reserved for Software (SW). These bits are reserved for software use and can be read and written at any time by the software, regardless of the current mode. The value of these bits is ignored by the hardware. The software field is set to zero at the initiation of either an exception or an interrupt.
2	Interrupt pending (IP). Set when an interrupt (software or hardware but NOT AST) is initiated; indicates an interrupt is in progress.
4:3	Current mode (CM). The access mode of the currently executing process as follows: 0 - Kernel 1 - Executive 2 - Supervisor 3 - User
6:5	Reserved to Digital, MBZ.
7	Virtual machine monitor (VMM) - When set, the processor is executing in a virtual machine monitor. When clear, the processor is running in either real or virtual machine mode. **PROGRAMMING NOTE** This bit is only meaningful when running with PALcode that implements virtual machine capabilities.
12:8	Interrupt priority level (IPL) - The current processor priority, in the range 0 to 31.
55:13	Reserved to Digital, MBZ.
61:56	Stack alignment (SP_ALIGN) - The previous stack byte alignment within a 64 byte aligned area, in the range 0 to 63. This field is set in the saved PS during the act of taking an exception or interrupt; it is used by the CALL_PAL REI instruction to restore the previous stack byte alignment.
63:62	Reserved to Digitial, MBZ.

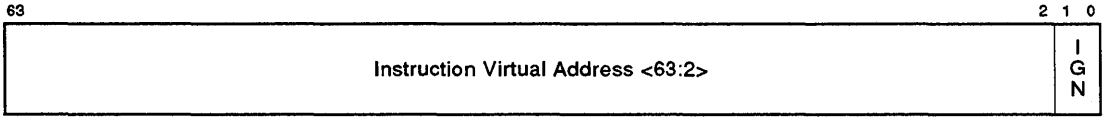
At bootstrap, the initial value of PS is set to $1F00_{16}$. Previous stack alignment is zero, IPL is 31, VMM is clear, CM is Kernel, and the SW and IP fields are zero.

6.2.2 Program Counter

The PC is a 64-bit virtual address. All instructions are aligned on longword boundaries and, therefore, hardware can assume zero for the two low-order PC bits.

The PC can be explicitly read with the Unconditional Branch (BR) instruction. All branching instructions also load a new value into the PC.

Figure 6–3: Program Counter (PC)



6.2.3 Processor Interrupt Priority Level (IPL)

Each processor has 32 interrupt priority levels (IPLs) divided into 16 software levels (numbered 0 to 15), and 16 hardware levels (numbered 16 to 31). User applications and most operating system software run at IPL 0, which may be thought of as process level. Higher numbered interrupt levels have higher priority; i.e., any request at an interrupt level higher than the processor's current IPL will interrupt immediately, but requests at lower or equal levels are deferred.

Interrupt levels 0 to 15 exist solely for use by software. No hardware event can request an interrupt on these levels. Conversely, interrupt levels 16 to 31 exist solely for use by hardware. Serious system failures, such as a machine check abort, however, raise the IPL to the highest level (31), to minimize processor interruption until the problem is corrected, and execute in Kernel mode on the Kernel stack.

6.2.4 Protection Modes

Each processor has four protection modes. The modes are Kernel, Executive, Supervisor, and User. Per-page memory protection varies as a function of mode (for example, a page can be made read-only in User mode, but read-write in Supervisor, Executive, or Kernel mode).

For each process, there is a separate stack associated with each mode. Corruption of one stack does not affect use of the other stacks.

Some instructions, termed privileged instructions, may only be executed in Kernel mode.

6.2.5 Processor Stacks

Each processor has four stacks. There are four process-specific stacks associated with the four modes of the current process. At any given time, only one of these stacks is actively used as the current stack.

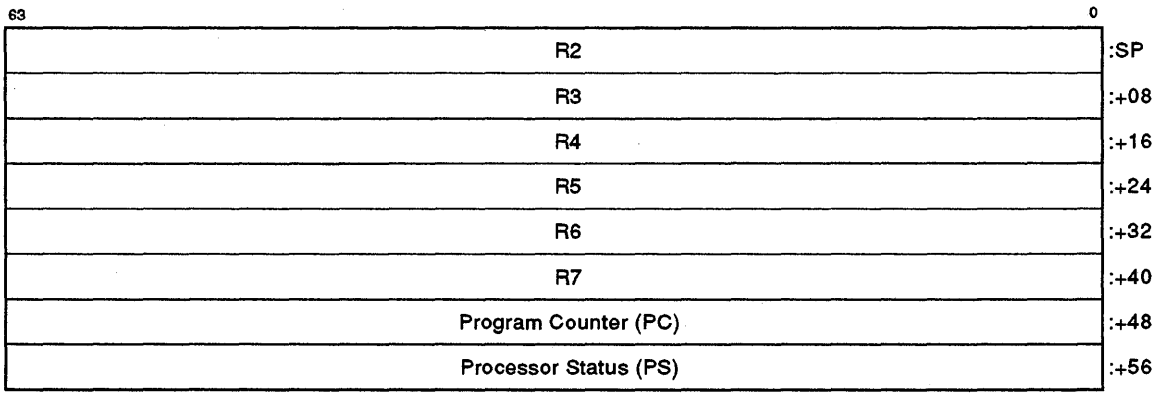
6.2.6 Stack Frames

When an exception, interrupt, or machine check occurs, a stack frame is pushed on the target stack. Regardless of the type of event notification, this stack frame consists of a 64 byte-aligned structure containing the saved contents of registers R2..R7, the Program Counter (PC), and the Processor Status (PS). Registers R2 and R3 are then loaded with vector and parameter from the SCB for the exception, interrupt, or machine check. Registers R4 and R5 may be loaded with data pertaining to the exception, interrupt, or machine check. The specific data loaded is described below in conjunction with each exception, interrupt, or machine check; if

no specific data is specified, the contents of R4 and R5 are UNPREDICTABLE. After the stack is built, the contents of registers R6 and R7 are UNPREDICTABLE.

The Program Counter value saved is that of the instruction encountering the exception in the case of faults, that of the next instruction in the case of traps and interrupts, and, on a best-effort basis, and that of the next instruction in the case of machine checks. Return from an exception, interrupt, or machine check is done via the CALL_PAL REI instruction, which restores the saved values of PC, PS, and R2..R7, thus re-executing the instruction in the case of faults, and proceeding to the next instruction in the case of traps, interrupts, and machine checks.

Figure 6–4: Stack Frame



6.3 Exceptions

Exception service routines execute in response to exception conditions caused by software. Most exception service routines execute in Kernel mode, on the Kernel stack; all exception service routines execute at the current processor IPL. Change Mode exception routines for CHMU/CHMS/CHME execute in the more privileged of the current mode or the target mode (U/S/E), on the matching stack. Exception service routines are usually coded to avoid exceptions; however, nested exceptions can occur.

There are three types of exceptions:

- A fault is an exception condition that occurs during an instruction and leaves the registers and memory in a consistent state such that elimination of the fault condition and subsequent re-execution of the instruction will give correct results. Faults are not guaranteed to leave the machine in exactly the same state it was in immediately prior to the fault, but rather in a state such that the instruction can be correctly executed if the fault condition is removed. The PC saved in the exception stack frame is the address of the faulting instruction. A CALL_PAL REI instruction to this PC will reexecute the faulting instruction.

- An arithmetic trap is an exception condition that occurs at the completion of the operation that caused the exception. Since several instructions may be in various stages of execution at any point in time, it is possible for multiple arithmetic traps to occur simultaneously. The PC that is saved in the exception frame on traps is that of the next instruction that would have been issued if the trapping condition(s) had not occurred. This is not necessarily the address of the instruction immediately following the one(s) encountering the trap condition, and intervening instructions may have changed operands or other state used by the instruction(s) encountering the trap condition(s). A CALL_PAL REI instruction to this PC will not reexecute the trapping instruction(s), nor will it reexecute any intervening instructions; it will simply continue execution from the point at which the trap was taken.

  In general, it is difficult to fixup results and continue program execution at the point of an arithmetic trap. Software can force a trap to be continued more easily without the need for complicated fixup code. This is accomplished by following a set of code-generation restrictions in code that could cause arithmetic traps which are to be completed by a software trap handler (see *Common Architecture, Chapter 4*), including specifying the /S software completion modifier in each such instruction.

  The AND of all the software completion modifiers for trapping instructions is provided to the arithmetic trap handler in the exception summary SWC bit. If SWC is set, a trap handler may find the trigger instruction by scanning backward from the trap PC until each register in the register write mask has been an instruction destination. The trigger instruction is the first instruction in I-stream order to get a trap within a trap shadow (see *Common Architecture, Chapter 4* for definition of trap shadow). If the SWC bit is clear, no fixup is possible (the trigger instruction may have been followed by a taken branch, so the trap PC cannot be used to find it).

- A synchronous trap is an exception condition that occurs at the completion of the operation that caused the exception (or, if the operation can only be partially carried out, at the completion of that part of the operation), and no subsequent instruction is issued before the trap occurs.

  Synchronous traps are divided into data alignment traps and all other synchronous traps.

6.3.1 Faults

The six types of faults signal that an instruction or its operands are in some way illegal. These faults are all initiated in Kernel mode and push an exception stack frame onto the stack. Upon entry to the exception routine, the saved PC (in the exception stack frame) is the virtual address of the faulting instruction.

The six faults include the Floating Disable Fault described in the next subsection and five memory management faults.

Memory management faults occur when a virtual address translation encounters an exception condition. This can occur as the result of instruction fetch or during a load or store operation.

Immediately following a memory management fault, register R4 contains the exact virtual address encountering the fault condition.

The register R5 contains the "MM Flag" quadword.

"MM Flag" is set as follows:

$0000\ 0000\ 0000\ 0000_{16}$ for a faulting data read

$0000\ 0000\ 0000\ 0001_{16}$ for a faulting I-fetch operation

$8000\ 0000\ 0000\ 0000_{16}$ for a faulting write operation

The faulting instruction is the instruction whose fetch faulted, or the load, store, or PALcode instruction that encountered the fault condition.

Chapter 3 describes the memory management architecture of Alpha in more detail.

6.3.1.1 Floating Disabled Fault

A Floating Disabled Fault is an exception that occurs when an attempt is made to execute a floating-point instruction and the floating enable (FEN) bit in the HWPCB is not set.

6.3.1.2 Access Control Violation (ACV) Fault

An ACV fault is a memory management fault indicating that an attempted access to a virtual address was not allowed in the current mode.

ACV faults usually indicate program errors, but in some cases, such as automatic stack expansion, can mean implicit operating system functions.

ACV faults take precedence over Translation Not Valid, Fault on Read, Fault on Write, and Fault on Execute faults.

ACV faults take precedence over Translation Not Valid faults so that a malicious user could not degrade system performance by causing spurious page faults to pages for which no access is allowed.

6.3.1.3 Translation Not Valid (TNV)

A TNV fault is a memory management fault that indicates that an attempted access was made to a virtual address whose Page Table Entry (PTE) was not valid.

Software may use TNV faults to implement virtual memory capabilities.

6.3.1.4 Fault On Read (FOR)

An FOR fault is a memory management fault that indicates that an attempted data read access was made to a virtual address whose Page Table Entry (PTE) had the Fault on Read bit set.

As a part of initiating the FOR fault, the processor invalidates the Translation Buffer entry that caused the fault to be generated.

IMPLEMENTATION NOTE

This allows an implementation only to invalidate entries from the Data-stream Translation Buffer on Fault On Read faults.

Note that the Translation Buffer may reload and cache the old PTE value between the time when the FOR fault invalidates the old value from the Translation Buffer and the time when software updates the PTE in memory. Software that depends on the processor-provided invalidate must thus be prepared to take another FOR fault on a page after clearing the page's PTE<FOR> bit. The second fault will invalidate the stale PTE from the Translation Buffer, and the processor cannot load another stale copy. Thus in the worst case, a multiprocessor system will take an initial FOR fault and then an additional FOR fault on each processor. In practice, even a single repetition is unlikely.

Software may use FOR faults to implement watchpoints, to collect page usage statistics, and to implement execute-only pages.

6.3.1.5 Fault On Write (FOW)

A FOW fault is a memory management fault that indicates that an attempted data write access was made to a virtual address whose Page Table Entry (PTE) had the Fault On Write bit set.

As a part of initiating the FOW fault, the processor invalidates the Translation Buffer entry that caused the fault to be generated.

IMPLEMENTATION NOTE

This allows an implementation only to invalidate entries from the Data-stream Translation Buffer on Fault On Write faults.

Note that the Translation Buffer may reload and cache the old PTE value between the time when the FOW fault invalidates the old value from the Translation Buffer and the time when software updates the PTE in memory. Software that depends on the processor-provided invalidate must thus be prepared to take another FOW fault on a page after clearing the page's PTE<FOW> bit. The second fault will invalidate the stale PTE from the Translation Buffer, and the processor cannot load another stale copy. Thus in the worst case, a multiprocessor system will take an initial FOW fault and then an additional FOW fault on each processor. In practice, even a single repetition is unlikely.

Software may use FOW faults to maintain modified page information, to implement copy on write and watchpoint capabilities, and to collect page usage statistics.

6.3.1.6 Fault On Execute (FOE)

An FOE fault is a memory management fault indicating that an attempted instruction stream access was made to a virtual address whose Page Table Entry (PTE) had the Fault On Execute bit set.

As a part of initiating the FOE fault, the processor invalidates the Translation Buffer entry that caused the fault to be generated.

IMPLEMENTATION NOTE

This allows an implementation only to invalidate entries from the Instruction-stream Translation Buffer on Fault On Execute faults.

Note that the Translation Buffer may reload and cache the old PTE value between the time when the FOE fault invalidates the old value from the Translation Buffer and the time when software updates the PTE in memory. Software that depends on the processor-provided invalidate must thus be prepared to take another FOE fault on a page after clearing the page's PTE<FOE> bit. The second fault will invalidate the stale PTE from the Translation Buffer, and the processor cannot load another stale copy. Thus in the worst case, a multiprocessor system will take an initial FOE fault and then an additional FOE fault on each processor. In practice, even a single repetition is unlikely.

Software may use FOE faults to implement access mode changes and protected entry to Kernel mode, to collect page usage statistics, and to detect programming errors that try to execute data.

6.3.2 Arithmetic Traps

An arithmetic trap is an exception that occurs as the result of performing an arithmetic or conversion operation.

If integer register R31 or floating register F31 is specified as the destination of an operation that can cause an arithmetic trap, it is UNPREDICTABLE whether the trap will actually occur, even if the operation would definitely produce an exceptional result.

Arithmetic traps are initiated in Kernel mode and push the exception stack frame on the Kernel stack. The Register Write Mask is saved in R4, and the Exception Summary parameter is saved in R5. These are described below.

When an arithmetic exception condition is detected, several instructions may be in various stages of execution. These instructions are allowed to complete before the arithmetic trap can be initiated. Some of these instructions may themselves cause further arithmetic traps. Thus it is possible for several arithmetic traps to be reported simultaneously.

It is also possible for the result of an instruction that causes an arithmetic trap to be used as an operand in a subsequent instruction before the trap is taken. If this would produce undesired behavior, software is responsible for inserting appropriate TRAPB instructions to cause the trap to be recognized before the result is used.

Integer exceptional results (integer overflow) can be forwarded to the address calculation for load and store instructions, to the address calculation for jump instructions, as the source data for a store instruction, or as the source data for a conditional branch instruction. This can result in the generation of an inappropriate address, the storing of exceptional results in memory, or an unintended branch.

If this would produce undesired behavior, software is responsible for inserting appropriate TRAPB instructions to cause the trap to be recognized before the result is used.

6.3.2.1 Exception Summary Parameter

The Exception Summary parameter records the various types of arithmetic traps that can occur together. These types of traps are described in subsections below.

Figure 6–5: Exception Summary

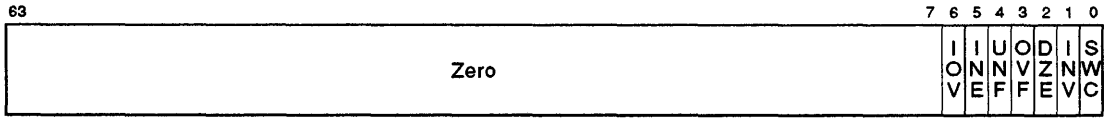


Table 6–3: Exception Summary

Bit	Description
0	Software Completion (SWC) Is set when all of the other arithmetic exception bits were set by floating-operate instructions with the /S software completion trap modifier set. See *Common Architecture, Chapter 4* for rules about setting the /S modifier in code that may cause an arithmetic trap, and Section 6.3 for rules about using the SWC bit in a trap handler.
1	Invalid Operation (INV) An attempt was made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values were illegal.
2	Division by Zero (DZE) An attempt was made to perform a floating divide operation with a divisor of zero.
3	Overflow (OVF) A floating arithmetic or conversion operation overflowed the destination exponent.
4	Underflow (UNF) A floating arithmetic or conversion operation underflowed the destination exponent.
5	Inexact Result (INE) A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.
6	Integer Overflow (IOV) An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision.

6.3.2.2 Register Write Mask

The Register Write Mask parameter records all registers that were targets of instructions that set the bits in the exception summary register. There is a one-to-one correspondence between bits in the Register Write Mask quadword and the register numbers. The quadword records, starting at bit 0 and proceeding right to left, which of the registers R0 through R31, then F0 through F31, received an exceptional result.

NOTE

For a sequence such as:

```
ADDF      F1,F2,F3
MULF      F4,F5,F3
```

if the add overflows and the multiply does not, the OVF bit is set in the exception summary, and the F3 bit is set in the register mask, even though the overflowed sum in F3 can be overwritten with an in-range product by the time the trap is taken. (This code violates the destination reuse rule for software completion. See *Common Architecture, Chapter 4* for the destination reuse rules.)

The PC value saved in the exception stack frame is the virtual address of the next instruction. This is defined as the virtual address of the first instruction not executed after the trap condition was recognized.

6.3.2.3 Invalid Operation (INV) Trap

An INV trap is reported for most floating-point operate instructions with an input operand that is a VAX reserved operand, VAX dirty zero, IEEE NaN, IEEE infinity, or IEEE denormal.

Floating INV traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.

6.3.2.4 Division by Zero (DZE) Trap

A DZE trap is reported when a finite number is divided by zero. Floating DZE traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.

6.3.2.5 Overflow (OVF) Trap

An OVF trap is reported when the destination's largest finite number is exceeded in magnitude by the rounded true result. Floating OVF traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.

6.3.2.6 Underflow (UNF) Trap

A UNF trap is reported when the destination's smallest finite number exceeds in magnitude the non-zero rounded true result. Floating UNF trap enable can be specified in each floating-point operate instruction. If underflow occurs, the result register is written with a true zero.

6.3.2.7 Inexact Result (INE) Trap

An INE trap is reported if the rounded result of an IEEE operation is not exact. INE trap enable can be specified in each IEEE floating-point operate instruction. The unchanged result value is stored in all cases.

6.3.2.8 Integer Overflow (IOV) Trap

An IOV trap is reported for any integer operation whose true result exceeds the destination register size. IOV trap enable can be specified in each arithmetic integer operate instruction and each floating-point convert-to-integer instruction. If integer overflow occurs, the result register is written with the truncated true result.

6.3.3 Synchronous Traps

A synchronous trap is an exception condition that occurs at the completion of the operation that caused the exception (or, if the operation can only be partially carried out, at the completion of that part of the operation), but no successor instruction is allowed to start. All traps that are not arithmetic traps are synchronous traps.

Some synchronous traps are caused by PALcode instructions: BPT, BUGCHK, CHMU, CHMS, CHME, and CHMK. For synchronous traps, the PC saved in the exception stack frame is the address of the instruction immediately following the one causing the trap condition. A CALL_PAL REI instruction to this PC will continue without reexecuting the trapping instruction. The following subsections describe the synchronous traps in detail.

6.3.3.1 Data Alignment Trap

All data must be naturally aligned or an alignment trap may be generated. Natural alignment means that data bytes are on byte boundaries, data words are on word boundaries, data longwords are on longword boundaries, and data quadwords are on quadword boundaries.

A Data Alignment trap is generated by the hardware when an attempt is made to load or store a longword or quadword to/from a register using an address that does not have the natural alignment of the particular data reference.

Data alignment traps are fixed up by the PALcode and are optionally reported to the operating system under the control of the DAT bit. If the bit is zero, the trap will be reported. If the bit is set, after the alignment is corrected, control is returned to the user. In either case, if the PALcode detects a LDx_L or STx_C instruction, no correction is possible and an illegal operand exception is generated.

The system software is notified via the generation of a Kernel mode exception through the Unaligned_Access SCB vector (280_{16}) The virtual address of the

unaligned data being accessed is stored in R4. R5 indicates whether the operation was a read or a write (0 = read/load 1 = write/store).

PALcode may write partial results to memory without probing to make sure all writes will succeed when dealing with unaligned store operations.

If a memory management exception condition occurs while reading or writing part of the unaligned data, the appropriate memory management fault is generated.

Software should avoid data misalignment whenever possible since the emulation performance penalty may be as large as 100 to 1.

The Data Alignment trap control bit is included in the HWPCB at offset +56 bit 63. In order to change this bit for the currently executing process, the DATFX IPR may be written via a CALL_PAL MTPR_DATFX instruction. This operation will also update the value in the HWPCB.

6.3.3.2 Other Synchronous Traps

With the traps described in this subsection, the SCB vector quadword is saved in R2 and the SCB parameter quadword is saved in R3. The change mode traps are initiated in the more privileged of the current mode and the target mode, while the other traps are initiated in Kernel mode.

6.3.3.2.1 Breakpoint Trap

A Breakpoint trap is an exception that occurs when a CALL_PAL BPT instruction is executed; see Chapter 2. Breakpoint traps are intended for use by debuggers and can be used to place breakpoints in a program.

Breakpoint traps are initiated in Kernel mode so that system debuggers can capture breakpoint traps that occur while the user is executing system code.

6.3.3.2.2 Bugcheck Trap

A Bugcheck trap is an exception that occurs when a CALL_PAL BUGCHK instruction is executed; see Chapter 2. Bugchecks are used to log errors detected by software.

6.3.3.2.3 Illegal Instruction Trap

An Illegal instruction Trap is an exception that occurs when an attempt is made to execute an instruction whose opcode is reserved to Digital, is a subsetted opcode that requires emulation on the host implementation, or is a privileged instruction and the current mode is not Kernel.

6.3.3.2.4 Illegal Operand Trap

An Illegal Operand Trap occurs when an attempt is made to execute PALcode with operand values that are illegal or reserved for future use by Digital.

Illegal operands include:

- An invalid combination of bits in the PS restored by the CALL_PAL REI instruction.

- An unaligned operand passed to PALcode.

6.3.3.2.5 Generate Software Trap

A Generate Software Trap is an exception that occurs when a CALL_PAL GENTRAP instruction is executed; see Chapter 2. The intended use is for low-level compiler-generated code that detects conditions such as divide-by-zero, range errors, subscript bounds and negative string lengths.

6.3.3.2.6 Change Mode to Kernel Trap

A Change Mode to Kernel trap is an exception that occurs when a CALL_PAL CHMK instruction is executed; see Chapter 2. Change Mode to Kernel traps are initiated in Kernel mode and push the exception frame on the Kernel stack.

6.3.3.2.7 Change Mode to Executive Trap

A Change Mode to Executive trap is an exception that occurs when a CALL_PAL CHME instruction is executed; see Chapter 2. Change Mode to Executive traps are initiated in the more privileged of the current mode and Executive mode, and push the exception frame on the target stack.

6.3.3.2.8 Change Mode to Supervisor Trap

A Change Mode to Supervisor trap is an exception that occurs when a CALL_PAL CHMS instruction is executed; see Chapter 2. Change Mode to Supervisor traps are initiated in the more privileged of the current mode and Supervisor mode, and push the exception frame on the target stack.

6.3.3.2.9 Change Mode to User Trap

A Change Mode to User trap is an exception that occurs when a CALL_PAL CHMU instruction is executed; see Chapter 2. Change Mode to User traps are initiated in the more privileged of the current mode and User mode, and push the exception frame on the target stack.

6.4 Interrupts

The processor arbitrates interrupt requests according to priority. When the priority of an interrupt request is higher than the current processor IPL, the processor will raise the IPL and service the interrupt request. The interrupt service routine is entered at the IPL of the interrupting source, in Kernel mode, and on the Kernel stack. Interrupt requests can come from I/O devices, memory controllers, other processors, or the processor itself.

The priority level of one processor does not affect the priority level of other processors. Thus, in a multiprocessor system, interrupt levels alone cannot be used to synchronize access to shared resources.

Synchronization with other processors in a multiprocessor system involves a combination of raising the IPL and executing an interlocking instruction sequence. Raising the IPL prevents the synchronization sequence itself from being interrupted on a single processor while the interlock sequence guarantees mutual exclusion with other processors. Alternately, one processor can issue explicit interprocessor

interrupts (and wait for acknowledgment) to put other processors in a known software state, thus achieving mutual exclusion.

In some implementations, several instructions may be in various stages of execution simultaneously. Before the processor can service an interrupt request, all active instructions must be allowed to complete without exception. Thus, when an exception occurs in a currently active instruction, the exception is initiated and the exception stack frame built immediately before the interrupt is initiated and its stack frame built.

The following events will cause an interrupt:

- Software interrupts - IPL 1 to 15.
- Asynchronous System Traps - IPL 2.
- Passive Release interrupts - IPL 20 to 23.
- I/O Device interrupts - IPL 20 to 23.
- Interval Clock interrupt - IPL 22.
- Interprocessor interrupt - IPL 22.
- Performance Monitor interrupt - IPL 29
- Powerfail interrupt - IPL 30.

Interrupts are initiated in Kernel mode and push the interrupt stack frame of eight quadwords onto the Kernel stack. The PC saved in the interrupt stack frame is the virtual address of the first instruction not executed after the interrupt condition was recognized. A CALL_PAL REI instruction to the saved PC/PS will continue execution at the point of interrupt.

Each interrupt source has a separate vector location (offset) within the System Control Block (SCB); see Section 6.6. With the exception of I/O device interrupts, each of the above events has a unique fixed vector. I/O device interrupts occupy a range of vectors that can be both statically and dynamically assigned. Upon entry to the interrupt service routine, R2 contains the SCB vector quadword and R3 contains the SCB parameter quadword. For Corrected Error interrupts, R4 optionally locates additional information; see Section 6.5.2.

In order to reduce interrupt overhead, no memory mapping information is changed when an interrupt occurs. Therefore, the instructions, data, and the contents of the interrupt vector for the interrupt service routine must be present in every process at the same virtual address.

Interrupt service routines should follow the discipline of not lowering IPL below their initial level. Lowering IPL in this way could result in an interrupt at an intermediate level which would cause the stack nesting to be incorrect.

Kernel mode software may need to raise and lower IPL during certain instruction sequences that must synchronize with possible interrupt conditions (such as powerfail). This can be accomplished by specifying the desired IPL and executing

a CALL_PAL MTPR_IPL instruction or by executing a CALL_PAL REI instruction that restores a PS that contains the desired IPL; see Chapter 2.

6.4.1 Software Interrupts - IPLs 1 to 15

6.4.1.1 Software Interrupt Summary Register

The architecture provides fifteen priority interrupt levels for use by software (level 0 is also available for use by software but interrupts can never occur at this level). The Software Interrupt Summary Register (SISR) stores a mask of pending software interrupts. Bit positions in this mask which contain a 1 correspond to the levels on which software interrupts are pending.

When the processor IPL drops below that of the highest requested software interrupt, a software interrupt is initiated and the corresponding bit in the SISR is cleared.

The SISR is a read-only internal processor register which may be read by Kernel mode software by executing a CALL_PAL MFPR_SISR instruction; see Section 5.3.

6.4.1.2 Software Interrupt Request Register

The Software Interrupt Request Register (SIRR) is a write-only internal processor register used for making software interrupt requests.

Kernel mode software may request a software interrupt at a particular level by executing a CALL_PAL MTPR_SIRR instruction; see Section 5.3.

If the requested interrupt level is greater than the current IPL, the interrupt will occur before the execution of the next instruction. If, however, the requested level is equal to or less than the current processor IPL, the interrupt request will be recorded in the Software Interrupt Summary Register (SISR) and deferred until the processor IPL drops to the appropriate level.

Note that no indication is given if there is already a request at the specified level. Therefore, the respective interrupt service routine must not assume that there is a one-to-one correspondence between interrupts requested and interrupts generated. A valid protocol for generating this correspondence is:

1. The requester places information in a control block and then inserts the control block in a queue associated with the respective software interrupt level.
2. The requester uses CALL_PAL MTPR_SIRR to request an interrupt at the appropriate level.
3. When enabling conditions arise, processor HW clears the appropriate SISR bit as part of initiating the software interrupt.
4. The interrupt service routine attempts to remove a control block from the request queue. If there are no control blocks in the queue, the interrupt is dismissed with a CALL_PAL REI instruction.
5. If a valid control block is removed from the queue, the requested service is performed and Step 3 is repeated.

6.4.2 Asynchronous System Trap - IPL 2

Asynchronous System Traps (ASTs) are a means of notifying a process of events that are not synchronized with its execution, but which must be dealt with in the context of the process. An AST is initiated in Kernel mode at IPL 2 when the current mode is less privileged than or equal to a mode for which an AST is pending and not disabled, with PS<IPL> less than 2; see Sections 6.7.6 and 4.3.

There are four separate per-mode SCB vectors, one for each of Kernel, Executive, Supervisor, and User modes.

On encountering an AST, the interrupt stack frame is pushed on the Kernel stack; the value of the PC saved in this stack frame is the address of the next instruction to have been executed if the interrupt had not occurred. The SCB vector quadword is saved in R2 and the SCB parameter quadword in R3.

6.4.3 Passive Release Interrupts—IPLs 20 to 23

Passive releases occur when the source of an interrupt granted by a processor cannot be determined. This can happen when the requesting I/O device determines that it no longer requires an interrupt after requesting one, or when a previously requested interrupt has already been serviced by another processor in some multiprocessor configurations. The interrupt handler for passive releases executes at the priority level of the interrupt request.

6.4.4 I/O Device Interrupts - IPLs 20 to 23

The architecture provides four priority levels for use by I/O devices. I/O device interrupts are requested when the device encounters a completion, attention, or error condition and the respective interrupt is enabled. \ See *Platform Section, Chapter 3* for more information. \

6.4.5 Interval Clock Interrupt - IPL 22

The Interval Clock requests an interrupt periodically.

At least 1000 interval clock interrupts occur per second. An entry in the HWRPB contains the number of interval clock interrupts per second that occur in an actual Alpha implementation, scaled up by 4096, and rounded to a 64-bit integer. \ (See *Platform Section, Chapter 3.*) \

The accuracy of the interval clock must be at least 50 parts per million (ppm).

HARDWARE/SOFTWARE NOTE

For example, an interval of 819.2 usec derived from a 10 MHz Ethernet clock and a 13-bit counter is acceptable.

To guarantee software progress, the interval clock interrupt should be no more frequent than the time it takes to do 500 main memory accesses. Over the life of the architecture, this interval may well decrease much more slowly than CPU cycle time decreases.

Other constraints may apply to Secure Kernel systems.

6.4.5.1 Interprocessor Interrupt - IPL 22

Interprocessor interrupts are provided to enable operating system software running on one processor to interrupt activity on another processor and cause operating system dependent actions to be performed.

6.4.5.1.1 Interprocessor Interrupt Request Register

The Interprocessor Interrupt Request Register (IPIR) is a write-only internal processor register used for making a request to interrupt a specific processor.

Kernel mode software may request to interrupt a particular processor by executing a CALL_PAL MTPR_IPIR instruction; see Section 5.3.

If the specified processor is the same as the current processor and the current IPL is less than 22, then the interrupt may be delayed and not initiated before the execution of the next instruction.

Note that, like software interrupts, no indication is given as to whether there is already an interprocessor interrupt pending when one is requested. Therefore, the interprocessor interrupt service routine must not assume there is a one-to-one correspondence between interrupts requested and interrupts generated. A valid protocol similar to the one for software interrupts for generating this correspondence is:

1. The requester places information in a control block and then inserts the control block in a queue associated with the target processor.
2. The requester uses CALL_PAL MTPR_IPIR to request an interprocessor interrupt on the target processor.
3. The interprocessor interrupt service routine on the target processor attempts to remove a control block from its request queue. If there are no control blocks remaining, the interrupt is dismissed with a CALL_PAL REI instruction.
4. If a valid control block is removed from the queue, the specified action is performed and Step 3 is repeated.

6.4.6 Performance Monitor Interrupts—IPL 29

These interrupts provide some of the support for processor or system performance measurements. The implementation is processor or system specific.

6.4.7 Powerfail Interrupt - IPL 30

If the system power supply backup option permits powerfail recovery, a Powerfail interrupt is generated to each processor when power is about to fail. \ See *Platform Section, Chapter 4* for a description of powerfail recovery requirements, and for a description of the interactions between system software and the console during system restarts. \

In systems in which the backup option maintains only the contents of memory and keeps system time with the BB_WATCH, the power supply requests a powerfail

interrupt to permit volatile system state to be saved. Prior to dispatching to the powerfail interrupt service routine, PALcode is responsible for saving all system state which is not visible to system software. Such state includes, but is not limited to, processor internal registers and PALcode temporary variables.

PALcode is also responsible for saving the contents of any writeback caches or buffers, including the powerfail interrupt stack frame. System software is responsible for saving all other system state. Such state includes, but is not limited to, processor registers and writeback cache contents. State can be saved by forcing all written data to a backed-up part of the memory subsystem; software may use the CALL_PAL CFLUSH instruction.

The Powerfail interrupt will not be initiated until the processor IPL drops below 30. Thus, critical code sequences can block the power-down sequence by raising the IPL to 31. Software, however, must take extra care not to lock out the power-down sequence for an extended period of time. \The time interval is platform specific.\

Explicit state is not provided by the architecture for software to directly determine whether there were outstanding interrupts when powerfail occurred. It is the responsibility of software to leave sufficient information in memory so that it may determine the proper action on power-up.

6.5 Machine Checks

A Machine Check, or mcheck, indicates that a hardware error condition was detected and may or may not be successfully corrected by hardware or PALcode. Such error conditions can occur either synchronously or asynchronously with respect to instruction execution. There are four types:

1. System Machine Check (IPL 31)

   These machine checks are generated by error conditions which are detected asynchronously to processor execution but are not successfully corrected by hardware or PALcode. Examples of system machine check conditions include protocol errors on the processor-memory-interconnect and unrecoverable memory errors.

   System machine checks are always maskable and deferred until processor IPL drops below IPL 31.

2. Processor Machine Check (IPL 31)

   These machine checks indicate that a processor internal error was detected and not successfully corrected by hardware or PALcode. Examples of processor machine check conditions include processor internal cache errors, translation buffer parity errors, or read access to a non-existent local I/O space location (NXM).

   Processor machine checks may be nonmaskable or maskable. If nonmaskable, they are initiated immediately, even if the processor IPL is 31. If maskable, they are deferred until processor IPL drops below IPL 31.

3. System Correctable Machine Check (IPL 20)

These machine checks are generated by error conditions that are detected asynchronously to processor execution and are successfully corrected by hardware or PALcode. Examples of system correctable machine check conditions include single bit errors within the memory subsystem.

System correctable machine checks are always maskable and deferred until processor IPL drops below IPL 20.

4. Processor Correctable Machine Check (IPL 31)

   These machine checks indicate that a processor internal error was detected and successfully corrected by hardware or PALcode. Examples of processor correctable machine check conditions include corrected processor internal cache errors and corrected translation buffer tab errors.

   Processor correctable machine checks may be nonmaskable or maskable. If nonmaskable, they are initiated immediately, even if the processor IPL is 31. If maskable, they are deferred until processor IPL drops below IPL 31.

Machine Checks are initiated in Kernel mode, on the Kernel stack, and cannot be disabled.

Correctable machine checks permit the pattern and frequency of certain errors to be captured. The delivery of these machine checks to system software can be disabled by setting IPR MCES<4:3>, as described in Chapter 5. Note that setting IPR MCES<4:3> does not disable the generation of the machine check or the correction of the error, but rather suppresses the reporting of that correction to system software.

The PC in the machine check stack frame is that of the next instruction that would have issued if the machine check condition had not occurred. This is not necessarily the address of the instruction immediately following the one encountering the error, and intervening instructions may have changed operands or other state used by the instruction encountering the error condition. A CALL_PAL REI instruction to this PC will simply continue execution from the point at which the machine check was taken.

NOTE

On machine checks, a meaningful PC is delivered on a best-effort basis. The machine state, processor registers, memory, and I/O devices may be indeterminate.

Machine checks may be deliberately generated by software, such as by probing non-existent-memory during memory sizing or searching for local I/O devices. In such a case, the DRAINA PALcode instruction can be called to force any outstanding machine checks to be taken before continuing.

6.5.1 Software Response

The reaction of system software to machine checks is specific to the characteristics of the processor, platform, and system software. System software must determine if operation should be discontinued on an implementation-specific basis.

To assist system software, PALcode provides a retry flag in the machine check logout frame (see Figure 6–6. If set, the state of the processor and platform hardware has not been compromised; system software operation should be able to continue.

If the retry flag is clear, the state of the processor is either unknown or is known to have been updated during partial execution of one or more instructions. System software operation can continue only after system software determines that the hardware state change permits and/or takes corrective action.

PALcode should take appropriate implementation-specific actions prior to setting the retry flag. PALcode should also attempt to ensure that each encountered error condition generates only one machine check.

IMPLEMENTATION NOTE

An important example of using the retry flag is read NXM.

Also, a read NXM should not generate both a Processor Machine Check and a System Machine Check.

PALcode sets an internal Machine-Check-In-Progress flag in the Machine Check Error Summary (MCES) register prior to initiating a system or processor machine check. System software must clear that flag to dismiss the machine check If a second uncorrectable machine check hardware error condition is detected while the flag is set, or if PALcode cannot deliver the machine check, PALcode forces the processor to enter console I/O mode, and subsequent actions, such as processor restart, are taken by the console. The REASON FOR HALT code is "double error abort encountered". \ See *Platform Section, Chapter 4.* \

Similiarly, PALcode sets an internal correctable Machine-Check-In-Progress flag in the Machine Check Error Summary (MCES) register prior to initiating a system correctable error interrupt or processor correctable machine check. System software must clear that flag to dismiss the condition and permit the reuse of the logout area. If a second correctable hardware error condition is detected while the flag is set, the error is corrected, but not reported. PALcode does not overwrite the logout area and the processor remains in program I/O mode.

6.5.2 Logout Areas

When a hardware error condition is encountered, PALcode optionally builds a logout frame prior to passing control to the machine check service routine. \ The logout frame is built in the Logout Area located by the processor's per-CPU slot in the HWRPB; see *Platform Section, Chapter 3.* \

Figure 6–6: Corrected Error and Machine Check Logout Frame

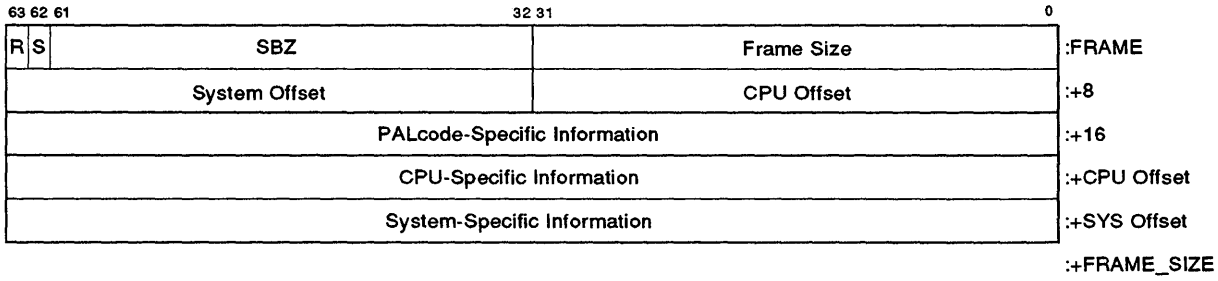


Table 6–4: Corrected Error and Machine Check Logout Frame Fields

Offset	Description
FRAME	FRAME SIZE - Size in bytes of the logout frame including the FRAME SIZE longword.
+04	FRAME FLAGS - Informational flags.

Bit	Description
31	RETRY FLAG - Indicates whether execution can be resumed after dismissing this machine check. Set on Corrected Error interrupts; may be set on Machine Checks.
30	SECOND ERROR FLAG - Indicates that a second correctable error was encountered. Set on Corrected Error interrupts when a correctable error was encountered while the relevant correctable error bit (PCE or SCE) is set in the MCES register. Clear on Machine Checks.
29–0	SBZ.

+08	CPU OFFSET - Offset in bytes from the base of the logout frame to the cpu-specific information. If 16 the frame contains no PALcode-specific information. If CPU OFFSET is equal to SYS OFFSET, the frame contains no cpu-specific information.

Table 6–4 (Cont.): Corrected Error and Machine Check Logout Frame Fields

Offset	Description
+12	SYS OFFSET - Offset in bytes from the base of the logout frame to the system-specific information. If SYS OFFSET is equal to FRAME SIZE, the frame contains no system-specific information.
+16	PALCODE INFORMATION - PALcode-specific logout information.
+CPU OFFSET	CPU INFORMATION - Cpu-specific logout information.
+SYS OFFSET	SYS INFORMATION - System platform-specific logout information.

The logout frame is optional; the service routine uses R4 to locate the frame, if any. Upon entry to the service routine, R4 contains the byte offset of the logout frame from the base of the logout area. If no frame was built, R4 contains –1 (FFFF FFFF FFFF $FFFF_{16}$).

6.6 System Control Block

The System Control Block (SCB) specifies the entry points for exception, interrupt, and machine check service routines. The block is from 8K to 32K bytes long, must be page aligned, and must be physically contiguous. The PFN is specified by the value of the System Control Block Base (SCBB) internal register.

The SCB consists of from 512 to 2048 entries, each 16 bytes long. The first 8 bytes of an entry, the vector, specify the virtual address of the service routine associated with that entry. The second 8 bytes, the parameter, are an arbitrary quadword value to be passed to the service routine.

The SCB entries are grouped into those for:

1. Faults
2. Arithmetic traps
3. Asynchronous system traps
4. Data alignment trap
5. Other synchronous traps
6. Processor software interrupts
7. Processor hardware interrupts
8. I/O device interrupts
9. Machine checks

The first 512 entries (offsets 0000 through $1FF0_{16}$) contain all architecturally defined and any statically allocated entries. All remaining SCB entries, if any, are used only for those I/O device interrupt vectors that are assigned dynamically by system software. It is the responsibility of that software to ensure the consistency of the assigned vector and the SCB entry.

6.6.1 SCB entries for faults

The exception handler for a fault executes with the IPL unchanged, in Kernel mode, on the Kernel stack.

Table 6–5: SCB Entries for Faults

Byte offset$_{16}$	Entry name
000	-unused-
010	Floating disabled fault
020-070	-unused-
080	Access Control Violation fault
090	Translation Not Valid fault
0A0	Fault on Read fault
0B0	Fault on Write fault
0C0	Fault on Execute fault
0A0-0F0	-unused-

6.6.2 SCB Entries for Arithmetic Traps

The exception handler for an arithmetic trap executes with the IPL unchanged, in Kernel mode, on the Kernel stack.

Table 6–6: SCB Entries for Arithmetic Traps

Byte offset$_{16}$	Entry name
200	Arithmetic Trap
210-230	-unused-

6.6.3 SCB Entries for Asynchronous System Traps (ASTs)

The interrupt handler for an asynchronous system trap executes at IPL 2, in Kernel mode, on the Kernel stack.

Table 6–7: SCB Entries for Asynchronous System Traps

Byte offset$_{16}$	Entry name
240	Kernel Mode AST
250	Executive Mode AST
260	Supervisor Mode AST

Table 6–7 (Cont.): SCB Entries for Asynchronous System Traps

Byte offset$_{16}$	Entry name
270	User Mode AST

6.6.4 SCB Entries for Data Alignment Traps

The exception handler for a data alignment trap executes with the IPL unchanged in Kernel mode, on the Kernel Stack.

Table 6–8: SCB Entries for Data Alignment Trap

Byte offset$_{16}$	Entry name
280	Unaligned_Access
290-3F0	-unused-

6.6.5 SCB Entries for other Synchronous Traps

The exception handler for a synchronous trap, other than those described above, executes with the IPL unchanged, in the mode and on the stack indicated below. "MostPriv" indicates that the handler executes in either the original mode or the new mode, whichever is the most privileged.

Table 6–9: SCB Entries for Other Synchronous Traps

Byte Offset$_{16}$	Entry Name	Mode
400	Breakpoint Trap	Kernel
410	Bug Check Trap	Kernel
420	Illegal Instruction Trap	Kernel
430	Illegal Operand Trap	Kernel
440	Generate Software Trap	Kernel
450	-unused-	
460	-unused-	
470	-unused-	
480	Change Mode to Kernel	Kernel
490	Change Mode to Executive	MostPriv
4A0	Change Mode to Supervisor	MostPriv
4B0	Change Mode to User	Current

Table 6–9 (Cont.): SCB Entries for Other Synchronous Traps

Byte Offset$_{16}$	Entry Name	Mode
4C0-4F0	-reserved for Digital-	

6.6.6 SCB Entries for Processor Software Interrupts

The exception handler for a processor software interrupt executes at the target IPL, in Kernel mode, on the Kernel stack.

Table 6–10: Entries for Processor Software Interrupts

Byte Offset$_{16}$	Entry Name	Target IPL$_{10}$
500	-unused-	
510	Software interrupt level 1	1
520	Software interrupt level 2	2
530	Software interrupt level 3	3
540	Software interrupt level 4	4
550	Software interrupt level 5	5
560	Software interrupt level 6	6
570	Software interrupt level 7	7
580	Software interrupt level 8	8
590	Software interrupt level 9	9
5A0	Software interrupt level 10	10
5B0	Software interrupt level 11	11
5C0	Software interrupt level 12	12
5D0	Software interrupt level 13	13
5E0	Software interrupt level 14	14
5F0	Software interrupt level 15	15

6.6.7 SCB Entries for Processor Hardware Interrupts

The interrupt handler for a processor hardware interrupt executes at the target IPL, in Kernel mode, on the Kernel stack.

Table 6–11: SCB Entries for Processor Hardware Interrupts

Byte Offset$_{16}$	Entry name	Target IPL$_{10}$
600	Interval clock interrupt	22
610	Interprocessor interrupt	22
640	Powerfail interrupt	30
650	Performance monitor	29
680-6E0	Reserved - processor specific	
6F0	Passive Release	20-23

Processor-specific SCB entries include those used by console devices (if any) or other peripherals dedicated to system support functions.

6.6.8 SCB Entries for I/O Device Interrupts

The interrupt handler for an I/O device interrupt executes at the target IPL, in Kernel mode, on the Kernel stack. SCB entries for offsets of 800_{16} through $7FF0_{16}$ are reserved for I/O device interrupts.

6.6.9 SCB Entries for Machine Checks

The handler for machine checks executes in Kernel mode, on the Kernel stack. The handler for system correctable machine checks executes at IPL 20; the handler for all other machine checks executes at IPL 31.

Table 6–12: SCB Entries for Machine Checks

Byte Offset$_{16}$	Entry Name	Target IPL$_{10}$
620	System correct. machine check	20
630	Processor correct. machine check	31
660	System machine check	31
670	Processor machine check	31

6.7 PALcode Support

6.7.1 Stack Writability

In response to various exceptions, interrupts, and machine checks, PALcode pushes information on the Kernel stack. PALcode may write this information without first probing to ensure that all such writes to the Kernel stack will succeed. If a memory management exception occurs while pushing information, PALcode forces the processor to enter console I/O mode, and subsequent actions, such as processor restart, are taken by the console. The REASON FOR HALT code is "processor halted due to kernel-stack-not-valid". \ See *Platform Section, Chapter 4.* \

6.7.2 Stack Residency

The User, Supervisor, and Executive stacks for the current process do not need to be resident. Software running in Kernel mode can bring in or allocate stack pages as TNV faults occur. However, since this activity is taking place in Kernel mode, the Kernel stack must be fully resident.

The faults TNV, ACV, FOR, and FOW, occurring on Kernel mode references to the Kernel stack, are considered serious system failures from which recovery is not possible. If any of these faults occur, PALcode forces the processor to enter console I/O mode, and subsequent actions, such as processor restart, are taken by the console. The REASON FOR HALT code is "processor halted due to kernel-stack-not-valid". \ See *Platform Section, Chapter 4.* \

6.7.3 Stack Alignment

Stacks may have arbitrary byte alignment, but performance may suffer if at least octaword alignment is not maintained by software.

PALcode creates stack frames in response to exceptions and interrupts. Before doing so, the target stack is aligned to a 64-byte boundary by setting the six low bits of the target SP to 000000_2. The previous value of these bits is stored in the SP_ALIGN field of the saved PS in memory, for use by a CALL_PAL REI instruction.

Software-constructed stack frames must be 64 byte aligned and have SP_ALIGN properly set; otherwise, a CALL_PAL REI instruction will take an illegal operand trap.

6.7.4 Initiate Exception or Interrupt or Machine Check

Exceptions and interrupts and machine checks are initiated by PALcode with interrupts disabled. When an exception, interrupt, or machine check, is initiated, the associated SCB vector is read to determine the address of the service routine. PALcode then attempts to push the PC, PS, and R2..R7 onto the target stack. When an interrupt (software or hardware but not AST) is initiated, PS<IP> is set to 1 to indicate an interrupt is in progress. Additional parameters may be passed in R4 and R5 on exceptions and machine checks.

During the attempt to push this information, the exceptions (faults) TNV, ACV, and FOW can occur:

- If any of those faults occur when the target stack is User, Supervisor, or Executive, then the fault is taken on the Kernel stack.
- If any of those faults occur when the target stack is the Kernel stack, PALcode forces the processor to enter console I/O mode, and subsequent actions, such as processor restart, are taken by the console. The REASON FOR HALT code is "processor halted due to kernel-stack-not-valid". \ See *Platform Section, Chapter 4.* \

6.7.5 Initiate Exception or Interrupt or Machine Check Model

```
check_for_exception_or_interrupt_or_mcheck:
   IF NOT {ready_to_initiate_exception OR
           ready_to_initiate_interrupt OR
           ready_to_initiate_mcheck} THEN
     BEGIN
       {fetch next instruction}
       {decode and execute instruction}
     END
   ELSE
     BEGIN
       {wait for instructions in progress to complete}
                     ! clear interrupt pending
       tmp ← 0

       IF {unmaskable mcheck pending} THEN
         BEGIN
           {back up implementation specific state if necessary}
           {attempt correction if appropriate}
           IF {uncorrectable AND MCES<0> = 1} THEN
               {enter console}
           ELSE IF {uncorrectable} THEN
               new_mode ← Kernel
               new_ipl ← 31
                              ! set mcheck error flag
               MCES<0> ← 1
           ELSE IF {reporting enabled} THEN
               new_mode ← Kernel
               new_ipl ← 31
               MCES<2> ← 1
           END
         END

       ELSE IF {data alignment trap} THEN
           new_mode ← Kernel

       ELSE IF {synchronous trap} THEN
         CASE {opcode} OF
           {back up implementation specific state if necessary}
           CHME: new_mode ← min(PS<CM>,Executive)
           CHMS: new_mode ← min(PS<CM>,Supervisor)
           CHMU: new_mode ← min(PS<CM>,User)
           otherwise: new_mode ← Kernel
         ENDCASE
```

```
ELSE IF {maskable uncorrectable mcheck pending and IPL < 31} THEN
  BEGIN
    {back up implementation specific state if necessary}
    IF {MCES<0> = 1} THEN
        {enter console}
    ELSE
        new_mode ←  Kernel
        new_ipl  ←  31
        MCES<0> ←  1 ! set mcheck error flag
    END
  END

ELSE
    new_mode ←  Kernel

END

IPR_SP[PS<CM>] ←  SP
new_sp ←  IPR_SP[new_mode]

IF {exception pending} THEN
  BEGIN
    {back up implementation specific state if necessary}
    new_ipl ←  PS<IPL>
  END

ELSE IF {interrupt pending} THEN
    new_ipl ←  {interrupt source IPL}
    tmp ←  1  ! set interrupt pending

ELSE IF {maskable correctable mcheck pending AND
         reporting enabled} THEN
     new_ipl ←  20
     MCES<1> ←  1
END

save_align ←  new_sp<5:0>
new_sp<5:0> ←  0

PUSH(PS OR LEFT_SHIFT(save_align,56), old_pc, new_mode)
PUSH(R7, R6, new_mode)
PUSH(R5, R4, new_mode)
PUSH(R3, R2, new_mode)

PS<SW> ←  0
PS<CM> ←  new_mode
PS<IP> ←  tmp
PS<IPL> ←  new_ipl
SP ←  new_sp

IF {memory management fault} THEN
    R4 ←  VA
    R5 ←  MMF
END

IF {data alignment trap} THEN
    R4 ←  VA
    R5 ←  { 0 if read/load  1 if write/store }
END
```

```
        IF {mcheck or correctable error interrupt} THEN
            IF {logout frame built}
               R4 ← logout_area_offset
            ELSE
               R4 ← -1
            END
        END

        IF {arithmetic Trap} THEN
            R4 ← register write mask
            R5 ← exception summary
        END

        IF {software interrupt} THEN
           SISR ← SISR AND NOT{ 2**{ PRIORITY_ENCODE(SISR) } }
        END

        vector ← {exception or interrupt or mcheck SCB offset}

        R2 ← (SCBB + vector)
        R3 ← (SCBB + vector + 8)
        PC ← R2

    END

    GOTO check_for_exception_or_interrupt_or_mcheck

PROCEDURE PUSH(first, last, mode)
    BEGIN
      IF ACCESS(new_sp - 16, mode) THEN
        BEGIN
          (new_sp - 8) ← first
          (new_sp - 16) ← last
          new_sp ← new_sp - 16
          RETURN
        END
      ELSE
          {initiate ACV, TNV, or FOW fault, or
          Kernel Stack Not Valid restart sequence}
      END
    END
```

6.7.6 PALcode Interrupt Arbitration

The following sections describe the logic for the interrupt conditions produced by the specified operation.

6.7.6.1 Writing the AST Summary Register

Writing the ASTSR internal processor register (see Section 5.3) requests an AST for any of the four processor modes. This may request an AST on a formerly inactive level and thus cause an AST interrupt.

The logic required to check for this condition is:

```
ASTSR<3:0> ← {ASTSR<3:0> AND R16<3:0>} OR R16<7:4>
IF ASTEN<0> AND ASTSR<0> AND {PS<IPL> LT 2} THEN
    {initiate AST interrupt at IPL 2}
```

6.7.6.2 Writing the AST Enable Register

Writing the ASTEN internal processor register (see Section 5.3) enables ASTs for any of the four processor modes. This may enable an AST on a formerly inactive level and thus cause an AST interrupt.

The logic required to check for this condition is:

```
ASTEN<3:0> ←  {ASTEN<3:0> AND R16<3:0>} OR R16<7:4>
IF ASTEN<0> AND ASTSR<0> AND {PS<IPL> LT 2} THEN
    {initiate AST interrupt at IPL 2}
```

6.7.6.3 Writing the IPL Register

Writing the IPL internal processor register (see Section 5.3) changes the current IPL. This may enable an AST or software interrupt on a formerly inactive level and thus cause an AST or software interrupt.

The logic required to check for this condition is:

```
PS<IPL> ←  R16<4:0>

! check for software interrupt at level 2..15

IF {RIGHT_SHIFT({SISR AND FFFC16 }, PS<IPL> + 1) NE 0} THEN
    {initiate software interrupt at IPL of high bit set in SISR}

! check for AST

IF ASTEN<0> AND ASTSR<0> AND {PS<IPL> LT 2} THEN
    {initiate AST interrupt at IPL 2}

! check for software interrupt at level 1

IF SISR<1> AND {PS<IPL> EQ 0} THEN
    {initiate software interrupt at IPL 1}
```

6.7.6.4 Writing the Software Interrupt Request Register

Writing the SIRR internal processor register (see Section 5.3) requests a software interrupt at one of the fifteen software interrupt levels. This may cause a formerly inactive level to cause a software interrupt.

The logic required to check for this condition is:

```
SISR<level> ←  1
IF level GT PS<IPL> THEN
    {initiate software interrupt at IPL level}
```

6.7.6.5 Return from Exception or Interrupt

The CALL_PAL REI instruction (see Chapter 2) writes both the Current Mode and IPL fields of the PS; see Section 6.2. This may enable a formerly disabled AST or software interrupt to occur.

The logic required to check for this condition is:

```
PS ←  New PS

! check for software interrupt at level 2..15
```

```
IF {RIGHT_SHIFT({SISR AND FFFC16 }, PS<IPL> + 1) NE 0} THEN
   {initiate software interrupt at IPL of high bit set in SISR}

! check for AST

tmp ← NOT LEFT_SHIFT(1110(bin), PS<CM>)
IF {{tmp AND ASTEN AND ASTSR}<3:0> NE 0} AND {PS<IPL> LT 2} THEN
    {initiate AST interrupt at IPL 2}

! check for software interrupt at level 1

IF SISR<1> AND {PS<IPL> EQ 0} THEN
    {initiate software interrupt at IPL 1}
```

6.7.6.6 Swap AST Enable

Swapping the AST enable state for the Current Mode results in writing the ASTEN internal processor register (see Section 5.3). This may enable a formerly disabled AST to cause an AST interrupt.

The logic required to check for this condition is:

```
R0 ← ZEXT(ASTEN<PS<CM>>)
ASTEN<PS<CM>> ← R16<0>

IF ASTEN<PS<CM>> AND ASTSR<PS<CM>> AND {PS<IPL> LT 2} THEN
    {initiate AST interrupt at IPL 2}
```

6.7.7 Processor State Transition Table

Table 6–13 shows the operations that can produce a state transition and the specific transition produced. For example, if a processor's initial state is Supervisor mode, it is not possible for the processor to transition to a program halt condition. A processor can only transition to program halt from Kernel mode.

In Table 6–13:

- *REI* increases mode or lowers IPL.
- *MTPR* changes IPL, or is a CALL_PAL MTPR_ASTSR or CALL_PAL MTPR_ASTEN instruction that causes an interrupt request.
- *Exc* is a state change caused by an exception.
- *Int* is a state change caused by an interrupt.
- *Mcheck* is a state change caused by a machine check.

Table 6–13: Processor State Transitions

Initial State:	Final State:				
	User	**Super.**	**Exec.**	**Kernel**	**Program Halt**
User	CHMU REI	CHMS	CHME	CHMK Exc Int Mcheck SWASTEN	Not Possible
Supervisor	REI	CHMS REI	CHME	CHMK Exc Int Mcheck SWASTEN	Not Possible
Executive	REI	REI	CHME REI	CHMK Exc Int Mcheck SWASTEN	Not Possible
Kernel	REI	REI	REI	CHMK REI Int Exc Mcheck MTPR SWASTEN	HALT

6.8 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Removed intr_flag and lock_flag from initiate excep inter mcheck model
2. Added eco #45—correctable errors (machine checks), performance monitor, and passive release information
3. Conditionalized references to platform section
4. Widget –> device
5. Reordered and combined sections to consolidate information
6. Added eco #30, #44 (DATFX) also eco #29 (GENTRAP)
7. Corrected init exception model for eco 25 PS(IP) bit and eco 23 (timer)
8. DRAINT to TRAPB
9. Converted to SDML
10. Added ECO #18, #23 (removed AT references), #25
11. Integrate references for Console ECO #15

Revision 4.0, March 29, 1991

1. On Memory Management Faults, R4 now contains the exact faulting address
2. Removed references to D_float
3. Typos
4. Note reason for unaligned load locked and store conditional vectors
5. Correct reference from AST Request Register to AST Summary Register
6. Correct pointer to location of physical address of error logout area from R2 to R4 in Processor Machine Check Abort section
7. Correct two references from Corrected Error logout area to Machine Check logout area
8. Change name of 'instruction issue model' to 'initiate exception or interrupt model'
9. Swap order of data alignment trap and synchronous trap code fragments in initiate exception or interrupt model
10. Correct which bits are loaded (=<4:0>) from R16 into IPR IPL by MTPR IPL
11. Add REI* and CHMx to each entry along the main diagonal of the Processor State Transition table
12. Describe machine check logout area as reserved for PALcode and console use
13. Add R2..R7 to values restored by REI in 'Stack Frames' text

14. Modify logic statement for Swap AST Enable so that it reflects CALL_PAL SWASTEN instruction action
15. Remove references to ASTs as 'interrupts', substituting 'exception' where appropriate
16. Move and modify reference to ASTs in last bullet item of section Exceptions to section Asynchronous System Trap
17. Define meaning of 'trigger instruction' in text of arithmetic trap description
18. Change values defined in R5 for memory management faults to full quadword values
19. Modify initiate exception or interrupt model to show bit corresponding to software interrupt being dispatched to is cleared before the dispatch
20. Modified tense of description of saved PC for arithmetic trap from 'would have issued' to 'would have been issued'
21. Move power-fail text at end of section Interprocessor Interrupt Request Register to end of subtext of section Interrupts
22. Clarify reference to 'RA' in initiate exception or interrupt pseudocode
23. Change 'vector ← {exception ..}' to 'vector ← {exception or interrupt ..}' in initiate exception or interrupt pseudocode
24. Note that there are four per-mode SCB vectors for ASTs
25. Add entry for Software Interrupts to table Exceptions and Interrupts Summary
26. Restrict the class of instructions that are described as taking Invalid Operation traps on non-finite values
27. Clarify that, following a memory management fault, R4 contains an address within the implementation-dependent-sized page that contains the faulting address
28. Reorganize the sections on synchronous traps (starting around current section $$section(synchr_trap)) to eliminate references to ASTs under Other Synchronous Traps category
29. Elaborate Interval Clock Interrupt description
30. Changed Load and Store D to G in SCB entries table for Alignment Traps
31. Moved 'perf. monitor' from Asynchronous Traps to Hardware Interrupts

Revision 3.0, March 2, 1990

1. Get PS/PC in correct order in stack frames
2. Restructure stack frames and R2..R7
3. Increase stack frame alignment to 64 byte
4. Restructure SCB

5. Change some faults to synchronous traps
6. Redo and simplify arithmetic traps
7. Rework AST delivery to match VAX
8. Specify writeback cache behavior at powerfail
9. Remove IPL from Processor State Transition Table
10. Remove Privileged instruction Trap

Revision 2.0, October 4, 1989

1. Remove interrupt stack
2. Remove kernel stack not valid abort
3. Remove stack alignment requirement, add PS<SP_ALIGN>
4. Remove ICIE and IPIE interrupt enables
5. Remove FREEZE of PC
6. Remove references to WAIT
7. Add DRAINT and DRAINA
8. Delete operand faults
9. Make data alignment fault stay in current mode
10. Simplify floating exceptions

Revision 1.0, May 23, 1989

1. First review distribution.

DEC OSF/1 Alpha Software (III)

This section describes how DEC OSF/1 operating system relates to the Alpha architecture, and includes the following chapters:

- Chapter 1, Introduction to DEC OSF/1 Alpha (III)
- Chapter 2, OSF/1 PALcode Instruction Descriptions (III)
- Chapter 3, OSF/1 Memory Management (III)
- Chapter 4, OSF/1 Process Structure (III)
- Chapter 5, OSF/1 Exceptions and Interrupts (III)

Contents

Chapter 3 OSF/1 Memory Management (III)

Chapter 4 OSF/1 Process Structure (III)

Chapter 5 OSF/1 Exceptions and Interrupts (III)

Figures

Tables

Chapter 1
Introduction to DEC OSF/1 Alpha (III)

The goals of this design are to provide a hardware implementation independent interface between the hardware and DEC OSF/1 Alpha. The interface needs to provide the needed abstractions to minimize the impact of different hardware implementations on the operating system. The interface also needs to be low in overhead to support high-performance systems. Lastly the interface needs to only support the features used by DEC OSF/1 Alpha.

The register usage in this interface is based on the current calling standard used by DEC OSF/1 Alpha. If the calling standard changes, this interface will be changed to reflect that. The current calling standard register usage is shown in Table 1–1.

Table 1–1: DEC OSF/1 Alpha Register Usage

Register Name	Software Name	Use and linkage
r0	v0	Used for expression evaluations and to hold integer function results.
r1..r8	t0..t7	Temporary registers; not preserved across procedure calls.
r9..r14	s0..s5	Saved registers; their values must be preserved across procedure calls.
r15	FP or s6	Frame pointer or a saved register.
r16..r21	a0..a5	Argument registers; used to pass the first 6 integer type arguments; their values are not preserved across procedure calls.
r22..r25	t8..t11	Temporary registers; not preserved across procedure calls.
r26	ra	Contains the return address; used for expression evaluation.
r27	pv or t12	Procedure value or a temporary register.
r28	at	Assembler temporary register; not preserved across procedure calls.
r29	GP	Global pointer.
r30	SP	Stack pointer.
r31	zero	Always has the value 0.

1.1 Programming Model

The programming model of the machine is the combination of the state visible either directly via instructions, or indirectly via actions of the machine. The following four tables define constants, state variables, terms, and subroutines used in the rest of the document.

1.1.1 Code Flow Constants

Table 1–2: Code Flow Constants

Term	Meaning and value
IPL = 2:0	The range 2:0 used in the PS to access the IPL field of the PS (PS<IPL>).
maxCPU	The maximum number of processors in a given system.
mode = 3	Used as a subscript in PS to select current mode (PS<mode>).
pageSize	Size of a page in an implementation in bytes.
vaSize	Size of virtual address in bits in a given implementation.

1.1.2 Machine State Terms

Table 1–3: Machine State Terms

Term	Meaning
ASN	An implementation-dependent size register to hold the current address space number (ASN). The size and existence of ASN is an implementation choice.
entArith<63:0>	The arithmetic trap entry address register. The entArith is an internal processor register that holds the dispatch address on an arithmetic trap. There can be a hardware register for the entArith or the PALcode can use private scratch memory.
entIF<63:0>	The instruction fault entry address register. The entIF is an internal processor register that holds the dispatch address on an instruction fault. There can be a hardware register for the entIF or the PALcode can use private scratch memory.
entInt<63:0>	The interrupt entry address register. The entInt is an internal processor register that holds the dispatch address on an interrupt. There can be a hardware register for the entInt or the PALcode can use private scratch memory.
entMM<63:0>	The memory-management fault entry address register. The entMM is an internal processor register that holds the dispatch address on a memory-management fault. There can be a hardware register for the entMM or the PALcode can use private scratch memory.

Table 1–3 (Cont.): Machine State Terms

Term	Meaning
entSys<63:0>	The system call entry address register. The entSys is an internal processor register that holds the dispatch address on an callsys instruction. There can be a hardware register for the entSys or the PALcode can use private scratch memory.
entUna<63:0>	The unaligned fault entry address register. The entUna is an internal processor register that holds the dispatch address on an unaligned fault. There can be a hardware register for the entUna or the PALcode can use private scratch memory.
FEN<0>	The floating-point enable register. The FEN is a one-bit register that is used to enable or disable floating-point instructions. If a floating-point instruction is executed with FEN equal to zero, a FEN fault is initiated.
instruction<31:0>	The current instruction being executed. This is a fake register used in the flows to CASE on different instructions.
intr_flag	A per-processor state bit. The intr_flag bit is cleared if that processor executes an rti or retsys instruction.
KGP<63:0>	The kernel global pointer. The KGP is an internal processor register that holds the kernel global pointer that is loaded into R15, the GP, when an exception is initiated. There can be a hardware register for the KGP or the PALcode can use private scratch memory.
KSP<63:0>	The kernel stack pointer. The KSP is an internal processor register that holds the kernel stack pointer while in user mode. There can be a hardware register for the KSP or the storage space in the PCB can be used.
lock_flag<0>	A one-bit register that is used by the load locked and store conditional instructions.
PC<63:0>	The program counter. The PC is a pointer to the next instruction in the flows. The low-order two bits of the PC always read as zero and writes to them are ignored.
PCB	The process control block. The PCB holds the state of the process.
PCBB<63:0>	The process control block base address register. The PCBB holds the address of the PCB for the current process.
PS<3:0>	The processor status. The PS is a four-bit register that stores the current mode in bit <3> and stores the three-bit IPL in bits <2:0>. The mode is 0 for kernel and 1 for user.
PTBR<63:0>	The page table base register. The PTBR contains the physical page frame number (PFN) of the highest level (level 1) page table.

Table 1–3 (Cont.): Machine State Terms

Term	Meaning
SP<63:0>	Another name for R30. The SP points to the top of the current stack. PALcode only accesses the kernel stack. The kernel stack must be quadword aligned whenever PALcode reads or writes it. If the PALcode accesses the kernel stack and the stack is not aligned, a kernel-stack-not-valid halt is initiated. Although PALcode does not access the user stack, that stack should also be at least quadword aligned for best performance.
sysvalue<63:0>	The system value register. The sysvalue holds the per-processor unique value. There can be a hardware register for the sysvalue register or the storage space in the PALcode scratch memory can be used. The sysvalue register can only be accessed by kernel mode code and there is one sysvalue register per CPU.
unique<63:0>	The process unique value register. The unique register holds the per-process unique value. There can be a hardware register for the unique register or the storage space in the PCB can be used. The unique register can be accessed by both user and kernel code and there is one unique register per process.
USP<63:0>	The user stack pointer. The USP is an internal processor register that holds the user stack pointer while in kernel mode. There can be a hardware register for the USP or the storage space in the PCB can be used.
VPTPTR<63:0>	The virtual page table pointer. The VPTPTR holds the virtual address of the first level page table.
whami<63:0>	The processor number of the current processor. This number is in the range 0..maxCPU−1.

1.1.3 Code Flow Terms

Table 1–4: Code Flow Terms

Term	Meaning
opDec	An attempt was made to execute a reserved instruction or execute a privileged instruction in user mode.

1.2 \Revision History

Revision 1.0, May 12, 1992

- First review distribution

\

Chapter 2

OSF/1 PALcode Instruction Descriptions (III)

2.1 Unprivileged PALcode Instructions

Table 2–1 lists the OSF/1 PALcode unprivileged instruction mnemonics, names, and the environment from which they can be called:

Table 2–1: Unprivileged OSF/1 PALcode Instructions

Mnemonic	Name	Calling environment
bpt	Breakpoint trap	Kernel and user modes
bugchk	Bugcheck trap	Kernel and user modes
callsys	System call	User mode
gentrap	Generate trap	Kernel and user modes
imb	I-Stream memory barrier	Kernel and user modes Described in *Common Architecture, Chapter 6*
rdunique	Read unique	Kernel and user modes
wrunique	Write unique	Kernel and user modes

2.1.1 Breakpoint Trap

Format:

```
bpt                                    ! PALcode format
```

Operation:

```
temp ← PS
if (ps<mode> NE 0) then
        USP ← SP                       !  Mode is user so switch to kernel
        SP  ← KSP
        PS  ← 0
endif
SP ← SP - {6 * 8}
(SP+00) ← temp
(SP+08) ← PC
(SP+16) ← GP
(SP+24) ← a0
(SP+32) ← a1
(SP+40) ← a2
a0 ← 0
GP ← KGP
PC ← entIF
```

Exceptions:

Kernel stack not valid

Mnemonics:

bpt	Breakpoint trap

Description:

The breakpoint trap (bpt) instruction switches mode to kernel, builds a stackframe on the kernel stack, loads the GP with the KGP, loads a value of 0 into a0, and dispatches to the breakpoint code pointed to by the entIF register. The registers a1..a2 are UNPREDICTABLE on entry to the trap handler. The saved PC at (SP+08) is the address of the instruction following the trap instruction that caused the trap.

Notes:

- The opcode and function code for the bpt instruction are the same in the OpenVMS and the OSF/1 PALcode.

2.1.2 Bugcheck Trap

Format:

```
bugchk                                  ! PALcode format
```

Operation:

```
temp ← PS
if (PS<mode> NE 0) then
        USP ← SP                        !  Mode is user so switch to kernel
        SP  ← KSP
        PS  ← 0
endif
SP ← SP - {6 * 8}
(SP+00) ← temp
(SP+08) ← PC
(SP+16) ← GP
(SP+24) ← a0
(SP+32) ← a1
(SP+40) ← a2
a0 ← 1
GP ← KGP
PC ← entIF
```

Exceptions:

Kernel stack not valid

Mnemonics:

bugchk Bugcheck trap

Description:

The bugcheck trap (bugchk) instruction switches mode to kernel, builds a stackframe on the kernel stack, loads the GP with the KGP, loads a value of 1 into a0, and dispatches to the breakpoint code pointed to by the entIF register. The registers a1..a2 are UNPREDICTABLE on entry to the trap handler. The saved PC at (SP+08) is the address of the instruction following the trap instruction that caused the trap.

Notes:

- The opcode and function code for the bugchk instruction are the same in the OpenVMS and the OSF/1 PALcode.

2.1.3 System Call

Format:

```
callsys                                  ! PALcode format
```

Operation:

```
if (PS<mode> EQ 0)  then
        machineCheck
endif
USP ←  SP
SP  ←  KSP
PS  ←  0                                 ! Mode=kernel
SP  ←  SP - {6*8}
(SP+00) ←  8                             ! PS of mode=user, IPL=0
(SP+08) ←  PC
(SP+08) ←  GP
GP ←  KGP
PC ←  entSys
```

Exceptions:

Machine check—invalid kernel mode callsys

Kernel stack not valid

Mnemonics:

callsys	System call

Description:

The system call (callsys) instruction is supported only from user mode. (Issuing a callsys from kernel mode causes a machine check exception).

The callsys instruction switches mode to kernel and builds a callsys stack frame. The GP is loaded with the KGP. The exception then dispatches to the system call code pointed to by the entsys register. On entry to the callsys code, the scratch registers t8..t11 are UNPREDICTABLE.

2.1.4 Generate Trap

Format:

```
gentrap                                  ! PALcode format
```

Operation:

```
temp ← PS
if (PS<mode> NE 0) then
        USP ← SP                         !  Mode is user so switch to kernel
        SP  ← KSP
        PS  ← 0
endif
SP ← SP - {6 * 8}
(SP+00) ← temp
(SP+08) ← PC
(SP+16) ← GP
(SP+24) ← a0
(SP+32) ← a1
(SP+40) ← a2
a0 ← 2
GP ← KGP
PC ← entIF
```

Exceptions:

Kernel stack not valid

Mnemonics:

gentrap Generate trap

Description:

The generate trap (gentrap) instruction switches mode to kernel, builds a stackframe on the kernel stack, loads the GP with the KGP, loads a value of 2 into a0, and dispatches to the breakpoint code pointed to by the entIF register. The registers a1..a2 are UNPREDICTABLE on entry to the trap handler. The saved PC at (SP+08) is the address of the instruction following the trap instruction that caused the trap.

Notes:

- The opcode and function code for the gentrap instruction are the same in the OpenVMS and the OSF/1 PALcode.

2.1.5 Read Unique Value

Format:

```
rdunique                        ! PALcode format
```

Operation:

```
v0 ← unique
```

Exceptions:

None

Mnemonics:

rdunique Read unique value

Description:

The read unique value (rdunique) instruction returns the process unique value in v0. The write unique value (wrunique) instruction, described in Section 2.1.6, sets the process unique value register.

Notes:

- The opcode and function code for the rdunique instruction are the same in the OpenVMS and the OSF/1 PALcode.

2.1.6 Write Unique Value

Format:

```
wrunique                        ! PALcode format
```

Operation:

```
unique ← a0
```

Exceptions:

None

Mnemonics:

wrunique Write unique value

Description:

The write unique value (wrunique) instruction sets the process unique register to the value passed in a0. The read unique value (rdunique) instruction, described in Section 2.1.5, returns the process unique value.

Notes:

- The opcode and function code for the wrunique instruction are the same in the OpenVMS and the OSF/1 PALcode.

2.2 Privileged OSF/1 PALcode Instructions

The Privileged OSF/1 PALcode instructions provide an abstracted interface to control the privileged state of the machine.

Table 2–2: Privileged OSF/1 PALcode Instructions

Mnemonic	Name
halt	Halt the Processor Described in *Common Architecture, Chapter 6*
rdps	Read processor status
rdusp	Read user stack pointer
rdval	Read system value
retsys	Return from system call
rti	Return from trap, fault, or interrupt
swpctx	Swap process context
swpipl	Swap IPL
tbi	TB (translation buffer) invalidate
whami	Who am I
wrent	Write system entry address
wrfen	Write floating-point enable
wrkgp	Write kernal global pointer
wrvptptr	Write virtual page table pointer

2.2.1 Read Processor Status

Format:

```
rdps                                    ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
v0 ← PS
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

rdps Read processor status

Description:

The read processor status (rdps) instruction returns the PS in v0. On return from the rdps instruction, registers t0 and t8..t11 are UNPREDICTABLE.

2.2.2 Read User Stack Pointer

Format:

```
rdusp                                   ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
v0 ← USP
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

rdusp Read user stack pointer

Description:

The read user stack pointer (rdusp) instruction returns the user stack pointer in v0. The user stack pointer is written by the wrusp instruction, described in Section 2.2.13. On return from the rdusp instruction, registers t0 and t8..t11 are UNPREDICTABLE.

2.2.3 Read System Value

Format:

rdval !PALcode format

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
v0 ← sysvalue
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

rdval Read system value

Description:

The read system value (rdval) instruction returns the sysvalue in v0, allowing access to a 64-bit per-processor value for use by the operating system. On return from the rdval instruction, registers t0 and t8..t11 are UNPREDICTABLE.

2.2.4 Return From System Call

Format:

```
retsys                                  ! PALcode format
```

Operation:

```
if {PS<mode> EQ 1} then
        {Initiate opDec fault}
endif
tmp ←   (SP+08)
GP  ←   (SP+16)
KSP ←   SP + {6*8}
SP  ←   USP
intr_flag = 0                   ! Clear the interrupt flag
lock_flag = 0                   ! Clear the load lock flag
PS  ←   8                        ! Mode=user
PC  ←   tmp
```

Exceptions:

Opcode reserved to Digital

Kernel stack not valid (halt)

Mnemonics:

retsys	Return from system call

Description:

The return from system call (retsys) instruction pops the return address and the user mode global pointer from the kernel stack. It then saves the kernel stack pointer, sets the mode to user, sets the IPL to zero, and enters the user mode code at the address popped off the stack.

2.2.5 Return From Trap, Fault or Interrupt

Format:

```
rti                                    ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
tempps ←  (SP+0)
temppc ←  (SP+8)
GP ←  (SP+16)
a0 ←  (SP+24)
a1 ←  (SP+32)
a2 ←  (SP+40)
SP ←  SP + {6 * 8}
if { tempps<3> EQ 1} then
        KSP ←  SP              !  New mode is user
        SP ←  USP
        tempps ←  8
endif
intr_flag = 0                  ! Clear the interrupt flag
lock_flag = 0                  ! Clear the load lock flag
PS ←  tempps<3:0>              ! Set new PS
PC ←  temppc
```

Exceptions:

Opcode reserved to Digital

Kernel stack not valid (halt)

Mnemonics:

rti Return from trap, fault, or interrupt

Description:

The return from fault, trap, or interrupt (rti) instruction pops registers (a0..a3, and GP), the PC, and the PS, from the kernel stack. If the new mode is user, the kernel stack is saved and the user stack is restored.

2.2.6 Swap Process Context

Format:

```
swpctx                                          ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1)
        {Initiate opDec fault}
endif
(PCBB) ←  SP                                    ! Save current state
(PCBB+8) ←  USP
tmp ←  PCC
tmp1 ←  tmp<31:0> + tmp<63:32>
(PCBB+24)<31:0> ←  tmp1<31:0>
v0 ←  PCBB                                      ! Return old PCBB
PCBB ←  a0                                      ! Switch PCBB
SP ←  (PCBB)                                    ! Restore new state
USP ←  (PCBB+8)
oldPTBR ←  PTBR
PTBR ←  (PCBB+16)
tmp1 ←  (PCBB+24)
PCC<63:32> ←  {tmp1 - tmp}<31:0>
FEN ←  (PCBB+40)
if {process unique register implemented} then
        (v0+32) ←  unique
        unique ←  (PCBB+32)
endif
if {ASN implemented}
        ASN ←  tmp1<63:32>
else
        if (oldPTBR  NE PTBR)
                {Invalidate all TB entries with ASM=0}
        endif
endif
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

swpctx Swap process context

Description:

The swap process context (swpctx) instruction saves the current process data in the current PCB. Then swpctx switches to the PCB passed in a0 and loads the

new process context. The old PCBB is returned in v0. On return from the swpctx instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

2.2.7 Swap IPL

Format:

```
swpipl                                  ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
v0 ← PS<IPL>
PS<IPL> ← a0<2:0>
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

swpipl Swap IPL

Description:

The swap IPL (swpipl) instruction returns the current value of the PS<IPL> bits in v0 and sets the IPL to the value passed in a0. On return from the spwipl instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

2.2.8 TB Invalidate

Format:

```
tbi                                    ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
case a0 begin
        1: ! tbisi
                {Invalidate ITB entry for va=a1}
                break;
        2: ! tbisd
                {Invalidate DTB entry for va=a1}
                break;
        3: ! tbis
                {Invalidate both ITB and DTB entry for va=a1}
                break;
       -1: ! tbiap
                {Invalidate all TB entries with ASM=0}
                break;
       -2: ! tbia
                {Flush all TBs}
                break;
       otherwise:
                break;
endcase
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

tbi TB (translation buffer) invalidate

Description:

The TB invalidate (tbi) instruction removes specified entries from the I and D translation buffers (TBs) when the mapping changes. The tbi instruction removes specific entry types based on a CASE selection of the value passed in register a0. On return from the tbi instruction, registers t0, t8..t11, a0, and a1 are UNPREDICTABLE.

2.2.9 Who Am I

Format:

```
whami                                   ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
          {Initiate opDec fault}
endif
v0 ←  whami
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

whami Who am I

Description:

The who am I (whami) instruction returns the processor number for the current processor in v0. The processor number is in the range 0 to the number of processors minus one (0..maxCPU–1) that can be configued in the system. On return from the whami instruction, registers t0 and t8..t11 are UNPREDICTABLE.

2.2.10 Write System Entry Address

Format:

```
wrent                                   ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
case a1 begin
        0:  ! Write the EntInt:
              entInt ← a0
              break;
        1:  ! Write the EntArith:
              entArith ← a0
              break;
        2:  ! Write the EntMM:
              entMM ← a0
              break;
        3: ! Write the EntIF:
              entIF ← a0
              break;
        4: ! Write the EntUna:
              entUna ← a0
              break;
        5: ! Write the EntSys:
              entSys ← a0
              break;
        otherwise:
              break;
endcase;
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

wrent	Write system entry address

Description:

The write system entry address (wrent) instruction determines the specific system entry point based on a CASE selection of the value passed in register a1. The wrent instruction then sets the virtual address of the specified system entry point to the value passed in a0.

For best performance all the addresses should be kseg addresses. (See Chapter 3 for a definition of kseg addresses).

On return from the wrent instruction, registers t0, t8..t11, a0, and a1 are UNPREDICTABLE.

2.2.11 Write Floating-Point Enable

Format:

```
wrfen                                   ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
FEN ←  a0<0>
(PCBB+40) ←  a0 AND 1
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

wrfen Write floating-point enable

Description:

The write floating-point enable (wrfen) instruction writes bit zero of the value passed in a0 to the floating-point enable register. The wrfen instruction also writes the value for FEN to the PCB at offset (PCBB+40). On return from the wrfen instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

2.2.12 Write Kernel Global Pointer

Format:

```
wrkgp                                   ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
KGP ← a0
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

wrkgp	Write kernal global pointer

Description:

The write kernel global pointer (wrkgp) instruction writes the value passed in a0 to the kernel global pointer (KGP) internal register. The KGP is used to load the GP on exceptions. On return from the wrkgp instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

2.2.13 Write User Stack Pointer

Format:

```
wrusp                                   ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
USP ← a0
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

wrusp Write user stack pointer

Description:

The write user stack pointer (wrusp) instruction writes the value passed in a0 to the user stack pointer. On return from the wrusp instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

2.2.14 Write System Value

Format:

```
wrval                                   !PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
sysvalue ←  a0
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

wrval	Write system value

Description:

The write system value (wrval) instruction writes the value passed in a0 to a 64-bit system value register. The combination of wrval with the rdval instruction, described in Section 2.2.3, allows access by the operating system to a 64-bit per-processor value. On return from the wrval instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

2.2.15 Write Virtual Page Table Pointer

Format:

```
wrvptptr                                ! PALcode format
```

Operation:

```
if (PS<mode> EQ 1) then
        {Initiate opDec fault}
endif
VPTPTR ←  a0
```

Exceptions:

Opcode reserved to Digital

Mnemonics:

wrvptptr Write virtual page table pointer

Description:

The write virtual page table pointer (wrvptptr) instruction writes the pointer passed in a0 to the virtual page table pointer register (VPTPTR). The VPTPTR is described in Chapter 3. On return from the wrvptptr instruction, registers t0, t8..t11, and a0 are UNPREDICTABLE.

2.3 \Revision History

Revision 1.0, May 12, 1992

- First review distribution

\

Chapter 3

OSF/1 Memory Management (III)

3.1 Introduction

3.2 Virtual Address Spaces

A virtual address is a 64-bit unsigned integer that specifies a byte location within the virtual address space. Implementations subset the supported address space to one of four sizes (43, 47, 51, or 55 bits) as a function of page size. The minimal supported virtual address size is 43 bits. If an implementation supports less than 64-bit virtual addresses, it must check that all the VA<63:vaSize> bits are equal to VA<vaSize–1>. This gives two disjoint ranges for valid virtual addresses. For example, for a 43-bit virtual address space, valid virtual address ranges are $0..3FFFFFFFFF_{16}$ and $FFFFFC0000000000_{16}..FFFFFFFFFFFFFFFF_{16}$. Access to virtual addresses outside of an implementation's valid virtual address range cause an access-violation fault.

The virtual address space is divided into 3 segments. The two bits va<vaSize–1:vaSize–2> select a segment as shown in Table 3–1.

Table 3–1: Virtual Address Space Segments

VA<vaSize–1:vaSize–2>	Name	Mapping	Access Control
0x	seg0	Mapped via TB	Programed in PTE
10	kseg	PA ← sext(VA<vaSize–3:0>)	Kernel Read/Write
11	seg1	Mapped via TB	Programed in PTE

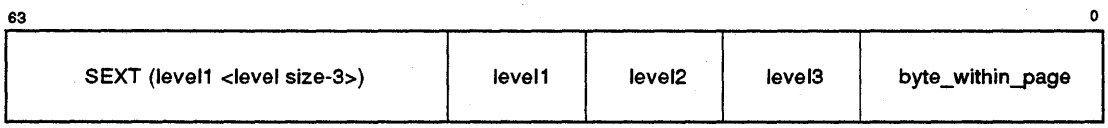
For kseg, the relocation, sharing, and protection are fixed. For seg0 and seg1, the virtual address space is broken into pages, which are the units of relocation, sharing, and protection. The page size ranges from 8 Kbytes to 64 Kbytes. Therefore, system software should allocate regions with differing protection on 64 Kbyte virtual address boundaries to ensure image compatibility across all Alpha implementations.

Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. The operating system controls the virtual-to-physical address mapping tables and saves the inactive (but used) parts of the virtual address space on external storage media.

3.2.1 Segment Seg0 and Seg1 Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. A seg0 or seg1 virtual address consists of three level-number fields and a byte_within_page field, as shown in Figure 3–1.

Figure 3–1: Virtual Address Format



The byte_within_page field can be either 13, 14, 15, or 16 bits depending on a particular implementation. Thus, the allowable page sizes are 8 Kbytes, 16 Kbytes, 32 Kbytes, and 64 Kbytes. Each level-number field is 0-n bits long, where, for example, *n* is 9 for an 8K page size. Level-number fields are the same size for a given implementation.

The level-number fields are a function of the page size; all page table entries at any given level do not exceed one page. The PFN field in the PTE is always 32 bits wide. Thus as the page size grows the virtual and physical address size also grows.

In Table 3–2, the physical address column is the maximum physical address supported by the smaller of seg0/seg1 or kseg, as indicated.

Table 3–2: Virtual Address Options

Page Size (bytes)	Byte Offset (bits)	Level Size (bits)	Virtual Address (bits)	Physical Address (bits)	Physical Address Limited by
8K	13	10	43	41	kseg
16K	14	11	47	45	kseg
32K	15	12	51	47	seg0/seg1
64K	16	13	55	48	seg0/seg1

3.2.2 Kseg Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. A kseg virtual address consists of segment select field with a value of 10_2 and a physical address field. The segment select field is the two bits va<vaSize–1:vaSize–2>. The physical address field is va<vaSize–3:0>.

Figure 3–2: Kseg Virtual Address Format

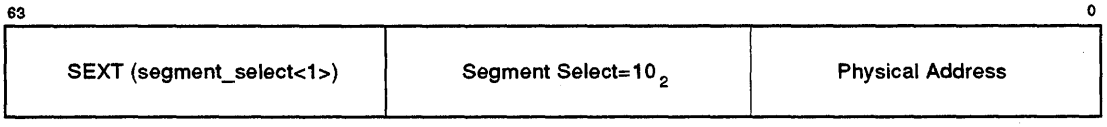
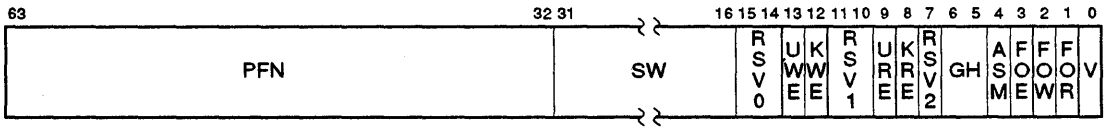


Figure 3–3: Page Table Entry (PTE)



3.3 Physical Address Space

Physical addresses are at most vaSize–2 bits. This allows all of physical memory to be accessed via kseg. A processor may choose to implement a smaller physical address space by not implementing some number of high order bits. The two most significant implemented physical address bits select a caching policy or implementation dependent type of address space. Implementations will use these bits as appropriate for their systems. For example, in a workstation with a 30-bit physical address space, bit<29> might select between memory and non-memory like regions, and bit <28> could enable or disable caching; see *Common Architecture, Chapter 5*.

3.4 Memory Management Control

Memory management is always enabled. Implementations must provide an environment for PALcode to service exceptions and to initialize and boot the processor. For example PALcode might run with I-stream mapping disabled.

3.5 Page Table Entries

The processor uses a quadword page table entry (PTE) to translate seg0 and seg1 virtual addresses to physical addresses. A PTE contains hardware and software control information and the physical page frame number (PFN). A PTE is a quadword with the following fields:

Table 3–3: Page Table Entry (PTE) Bit Summary

Bits	Name	Meaning
63:32	PFN	Page frame number The PFN field always points to a page boundary. If V is set, the PFN is concatenated with the byte_within_page bits of the virtual address to obtain the physical address.
31:16	SW	Reserved for software.
15:14	RSV0	Reserved for hardware; SBZ.
13	UWE	User write enable. This bit enables writes from user mode. If this bit is 0 and a store is attempted while in user mode, an access-violation fault occurs. This bit is valid even when V=0.
12	KWE	Kernel write enable. This bit enables writes from kernel mode. If this bit is 0 and a store is attempted while in kernel mode, an access-violation fault occurs. This bit is valid even when V=0.
11:10	RSV1	Reserved for hardware; SBZ.
9	URE	User read enable. This bit enables reads from user mode. If this bit is 0 and a load or instruction fetch is attempted while in user mode, an Access Violation occurs. This bit is valid even when V=0.
8	KRE	Kernel read enable. This bit enables reads from kernel mode. If this bit is 0 and a load or instruction fetch is attempted while in kernel mode, an access-violation fault occurs. This bit is valid even when V=0.
7	RSV2	Reserved for hardware; SBZ.
6:5	GH	Granularity hint. Software may set these bits to a non-zero value to supply a hint to translation buffer implementations that a block of pages can be treated as a single larger page: 1. A block is an aligned group of 8**N pages where N is the value of PTE<6:5>, e.f. a group of 1, 8, 64, or 512 pages starting at a virtual address with page_size + 3*N low-order zeros. 2. The block is a group of physically contiguous pages that are aligned both virtually and physically. Within the block, the low 3*N bits of the PFNs describe the identity mapping and the high 32–3*N PFN bits are all equal. 3. Within the block, all PTEs have the same values for bits <15:0>. Hardware may use this hint to map the entire block with a single TB entry, instead of 8, 64, or 512 separare TB entries.

Table 3–3 (Cont.): Page Table Entry (PTE) Bit Summary

Bits	Name	Meaning
4	ASM	Address space match. When set, this PTE matches all address space numbers. For a given VA, ASM must he set consistently in all processes, otherwise the address mapping is UNPREDICTABLE.
3	FOE	Fault on execute. When set, a Fault on Execute exception occurs on an attempt to execute any location in the page.
2	FOW	Fault on write. When set, a Fault on Write exception occurs on an attempt to write any location in the page.
1	FOR	Fault on read. When set, a Fault on Read exception occurs on an attempt to read any location in the page.
0	V	Valid. Indicates the validity of the PFN field. When V is set the PFN field is valid for use by hardware. When V is clear, the PFN field is reserved for use by software. The V bit does not affect the validity of PTE<15:1> bits.

3.5.1 Changes to Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, the operating system may set or clear the V bit, change the PFN field as pages are moved to and from external storage media, or modify the software bits. The processor hardware never changes PTEs.

Software must guarantee that each PTE is always consistent within itself. Changing a PTE one field at a time can cause incorrect system operation, such as setting PTE<V> with one instruction before establishing PTE<PFN> with another. Execution of an interrupt service routine between the two instructions could use an address that would map using the inconsistent PTE. Software can solve this problem by building a complete new PTE in a register and then moving the new PTE to the page table by using an STQ instruction.

Multiprocessing makes the problem more complicated. Another processor could be reading (or even changing) the same PTE that the first processor is changing. Such concurrent access must produce consistent results. Software must use some form of software synchronization to modify PTEs that are already valid. Whenever a processor modifies a valid PTE, it is possible that other processors in a multiprocessor system may have old copies of that PTE in their translation buffer. Software must inform other processors of changes to PTEs. Hardware must ensure that aligned quadword reads and writes are atomic operations. Hardware must not cache invalid PTEs (PTEs with the V bit equal to 0) in translation buffers. See Section 3.8 for more information.

3.6 Memory Protection

Memory protection is the function of validating whether a particular type of access is allowed to a specific page from a particular access mode. Access to each page is controlled by a protection code that specifies, for each access mode, whether read or write references are allowed. The processor uses the following to determine whether an intended access is allowed:

- The virtual address, which is used to either select kseg mapping or provide the index into the page tables.
- The intended access type (read or write).
- The current access mode base on Processor Mode.

For protection checks, the intended access is read for data loads and instruction fetches, and write for data stores.

3.6.1 Processor Access Modes

There are two processor modes, user and kernel. The access mode of a running process is stored in the processor status mode bit (PS<mode>).

3.6.2 Protection Code

Every page in the virtual address space is protected according to its use. A program may be prevented from reading or writing portions of its address space. Associated with each page is a protection code that describes the accessibility of the page for each processor mode.

For seg0 and seg1, the code allows a choice of read or write protection for each processor mode. For each mode, access can be read/write, read-only, or no-access. Read and write accessibility and the protection for each mode are specified independently.

For kseg, the protection code is kernel read/write, user no-access.

3.6.3 Access-Violation Faults

An access-violation memory-management fault occurs if an illegal access is attempted, as determined by the current processor mode and the page's protection.

3.7 Address Translation for Seg0 and Seg1

The page tables can be accessed from physical memory, or (to reduce overhead) can be mapped to a linear region of the virtual address space. The following sections describe both access methods.

3.7.1 Physical Access for Seg0 and Seg1 PTEs

Seg0 and seg1 address translation can be performed by accessing entries in a three-level page table structure. The page table base register (PTBR) contains the physical page frame number (PFN) of the highest level (level 1) page table. Bits <level1> of the virtual address are used to index into the first level page table to obtain the

physical PFN of the base of the second level (level 2) page table. Bits <level2> of the virtual address are used to index into the second level page table to obtain the physical PFN of the base of the third level (level 3) page table. Bits <level3> of the virtual address are used to index the third level page table to obtain the physical PFN of the page being referenced. The PFN is concatenated with virtual address bits <byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table does not reside in a memory-like region, or does reside in nonexistent memory, the operation of the processor is UNDEFINED.

If the first-level or second-level PTE is valid, the protection bits are ignored; the protection code in the third-level PTE is used to determine accessibility. If a first level or second level PTE is invalid, an access-violation fault occurs if the PTE<KRE> equals zero. An access-violation fault on a first-level or second-level PTE implies that all lower-level page tables mapped by that PTE do not exist.

The algorithm to generate a physical address from a seg0 or seg1 virtual address follows:

```
IF {SEXT(VA<vaSize-1:0>) neq VA} THEN
        { initiate access-violation fault}

level1_PTE ←   ({PTBR * page_size} + {8 * VA<level1>} )              ! Read physical
IF level1_PTE<v> EQ 0 THEN
        IF level1_PTE<KRE> eq 0 THEN
                { initiate access-violation fault}
        ELSE
                { initiate translation-not-valid fault}

level2_PTE ←   ({level1_PTE<PFN> * page_size} + {8 * VA<level2>} )    ! Read physical
IF level2_PTE<v> EQ 0 THEN
        IF level2_PTE<KRE> eq 0 THEN
                { initiate access-violation fault}
        ELSE
                { initiate translation-not-valid fault}

level3_PTE ←   ({level2_PTE<PFN> * page_size} + {8 * VA<level3>} )    ! Read physical

IF {{{level3_PTE<UWE> eq 0} AND {write access} AND {ps<mode> EQ 1} } OR
    {{level3_PTE<URE> eq 0} AND {read access} AND {ps<mode> EQ 1} } OR
    {{level3_PTE<KWE> eq 0} AND {write access} AND {ps<mode> EQ 0} } OR
    {{level3_PTE<KRE> eq 0} AND {read access} AND {ps<mode> EQ 0} } }
        THEN
                {initiate memory-management fault}
        ELSE
                IF level3_PTE<v> EQ 0 THEN
                        {initiate memory-management fault}

IF { level3_PTE<FOW> eq 1} AND {write access} THEN
        {initiate memory-management fault}
IF { level3_PTE<FOR> eq 1} AND {read access} THEN
        {initiate memory-management fault}
IF { level3_PTE<FOE> eq 1} AND {execute access} THEN
        {initiate memory-management fault}

Physical_address ←  {level3_PTE<PFN> * page_size} OR VA<byte_within_page>
```

3.7.2 Virtual Access for Seg0 or Seg1 PTEs

The page tables can be mapped into a linear region of the virtual address space, reducing the overhead for seg0 and seg1 PTE accesses. The mapping is done as follows:

1. Select a $2^{(3*lg(pageSize/8))+3}$ byte-aligned region (an address with $3*lg(pageSize/8)+3$ low-order zeros) in the seg0 or seg1 address space. Set the virtual page table pointer (VPTPTR) with a write virtual page table pointer instruction (wrvptptr) to the selected value.

2. Create a level1 PTE to map the page tables as follows.

```
level1_PTE = 0              ! Initialize all fields to 0
level1_PTE<63:32> = pfn_of_Level_1_pagetable
                            ! Set the PFN to the PFN of the level one pagetable
level1_PTE<8>  = 1          ! Set the kernel read enable bit
level1_PTE<0>  = 1          ! Set the valid bit
```

3. Set the level1 page table entry that corresponds to the VPTB to the created level1_PTE.

4. Set all level1 and level 2 valid PTEs to allow kernel read access. With this setup in place the algorithm to fetch a seg0 or seg1 PTE is:

```
tmp ←  left_shift (va, {64 - {{lg(pageSize) *4} - 9}} )
tmp ←  right_shift (tmp, {64 - {{lg(pageSize) *4} - 9} + lg(pageSize) - 3} )
tmp ←  VPTB OR tmp
tmp<2:0> ←  0
level3_PTE ←  (tmp)      ! Load PTE using it's virtual address
```

The virtual access method is used by PALcode for most TB fills.

3.8 Translation Buffer

In order to save actual memory references when repeatedly referencing the same pages, hardware implementations include a translation buffer to remember successful virtual address translations and page states. When the process context is changed, a new value is loaded into the address space number (ASN) internal processor register with a swap process context (swpctx) instruction. This causes address translations for pages with PTE<ASM> clear to be invalidated on a processor that does not implement address space numbers.

Additionally, when the software changes any part (except the software field) of a valid PTE, it must also execute a CALL_PAL tbi instruction. The entire translation buffer can be invalidated by tbia, and all ASM=0 entries can be invalidated by tbiap. The translation buffer must not store invalid PTEs. Therefore, the software is not required to invalidate translation buffer entries when making changes for PTEs that are already invalid.

3.9 Address Space Numbers

The Alpha architecture allows a processor to optionally implement address space numbers (process tags) to reduce the need for invalidation of cached address translations for process specific addresses when a context switch occurs. \
The supported address space number (ASN) range is 0..MAX_ASN, MAX_ASN is provided in the HWRPB MAX_ASN field. \

The address space number for the current process is loaded by software in the address space number (ASN) with a swpctx instruction. ASNs are processor specific and the hardware makes no attempt to maintain coherency across multiple

processors. In a multiprocessor system, software is responsible for ensuring the consistency of TB entries for processes that might be rescheduled on different processors.

\ Systems that support ASNs should have MAX_ASN in the range 13..65535. The number of ASNs should be determined by the market a system is targeting. \

PROGRAMMING NOTE

System software should not assume that the number of ASNs is a power of two. This allows, for example, hardware to use N TB tag bits to encode (2**N)–3 ASN values, one value for ASM=1 PTEs, and one for invalid.

There are several possible ways of using ASNs. There are several complications in a multiprocessor system. Consider the case where a process that executed on processor–1 is rescheduled on processor–2. If a page is deleted or its protection is changed, the TB in processor–1 has stale data. One solution would be to send an interprocessor interrupt to all the processors on which this process could have run and cause them to invalidate the changed PTE. This results in significant overhead in a system with several processors. Another solution would be to have software invalidate all TB entries for a process on a new processor before it can begin execution, if the process executed on another processor during its previous execution. This ensures the deletion of possibly stale TB entries on the new processor. A third solution would assign a new ASN whenever a process is run on a processor that is not the same as the last processor on which it ran.

3.10 Memory-Management Faults

On a memory-management fault, the fault code (MMCSR) is passed in a1 to specify the type of fault encountered, as shown in Table 3–4.

Table 3–4: Memory-Management Fault Type Codes

Fault	MMCSR value
Translation not valid	0
Access violation	1
Fault on read	2
Fault on execute	3

Table 3–4 (Cont.): Memory-Management Fault Type Codes

Fault	MMCSR value
Fault on write	4

- A translation-not-valid fault is taken when a read or write reference is attempted through an invalid PTE in a first, second, or third-level page table.
- An access-violation fault is taken on a reference to a seg0 or seg1 address when the protection field of the third-level PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode. An access-violation fault is also taken if the KRE bit is a zero in an invalid first or second level PTE. An access-violation fault is generated for any access to a kseg address when the mode is user (PS<mode> EQ 1).
- A fault-on-read (FOR) fault occurs when a read is attempted with PTE<FOR> set.
- A fault-on-execute (FOE) fault occurs when an instruction fetch is attempted with PTE<FOE> set.
- A fault-on-write (FOW) fault occurs when a write is attempted with PTE<FOW> set.

3.11 \Revision History

Revision 1.0, May 12, 1992

- First review distribution

\

Chapter 4

OSF/1 Process Structure (III)

4.1 Process Definition

A process is a single thread of execution. It is the basic entity that can be scheduled and is executed by the processor. A process consists of an address space and both software and hardware context. The hardware context of a process is defined by the the following:

- 30 integer registers (excluding R31 and SP)
- 31 floating-point registers (excluding F31)
- The program counter (PC)
- The two per-process stack pointers (USP/KSP)
- The processor status (PS)
- The address space number (ASN)
- The process cycle counter (PCC)
- The page table base register (PTBR)
- The process unique value (unique)

This information must be loaded if a process is to execute.

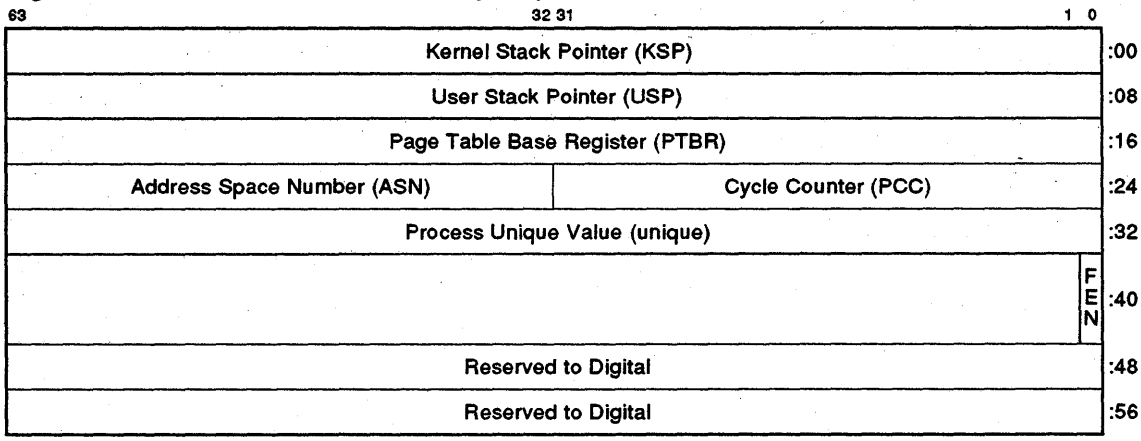
While a process is executing, some of its hardware context is being updated in the internal registers. When a process is not being executed, its hardware context is stored in memory in a software structure termed the process control block (PCB). Saving the process context in the PCB and loading new values from another PCB for a new context is termed context switching. Context switching occurs as one process after another is scheduled for execution.

4.2 Process Control Block (PCB)

As shown in Figure 4–1, the PCB holds the state of a process.

The contents of the PCB are loaded and saved by the swpctx instruction. The PCB must be quadword aligned and should be 64 byte aligned for best performance. Kernel mode code can read the PTBR, the ASN, and the FEN for the current process from the PCB. Kernel mode code must use the rdusp/wrusp instructions to access the USP. The PCC must be read with the rpcc instruction. The unique value can be accessed with the rdunique and wrunique instructions.

Figure 4–1: Process Control Block (PCB)



4.3 \Revision History

Revision 1.0, May 12, 1992

- First review distribution

\

Chapter 5

OSF/1 Exceptions and Interrupts (III)

5.1 Introduction

At certain times during the operation of a system, events within the system require the execution of software outside the explicit flow of control. When such an event occurs, an Alpha processor forces a change in control flow from that indicated by the current instruction stream. The notification process for such an event is either an exception or an interrupt.

5.1.1 Exceptions

Exceptions are relevant primarily to the currently executing process. Exception service routines execute in response to exception conditions caused by software. All exception service routines execute in kernel mode on the kernel stack. Exception conditions consist of faults, arithmetic traps, and synchronous traps:

- A fault occurs during an instruction and leaves the registers and memory in a consistent state such that elimination of the fault condition and subsequent reexecution of the instruction gives correct results. Faults are not guaranteed to leave the machine in exactly the same state it was in immediately prior to the fault, but rather in a state such that the instruction can be correctly executed if the fault condition is removed. The PC saved in the exception stack frame is the address of the faulting instruction. An rti instruction to that PC reexecutes the faulting instruction.

- An arithmetic trap occurs at the completion of the operation that caused the exception. Since several instructions may be in various stages of execution at any point in time, it is possible for multiple arithmetic traps to occur simultaneously.

  The PC that is saved in the exception frame on traps is that of the next instruction that would have been issued if the trapping conditions had not occurred. However, that PC is *not* necessarily the address of the instruction immediately following the instructions that encountered the trap condition. Further, intervening instructions may have changed operands or other state used by the instructions encountering the trap conditions.

  An rti instruction to that PC does not reexecute the trapping instructions, nor does it reexecute any intervening instructions; it simply continues execution from the point at which the trap was taken.

  In general, it is difficult to fix up results and continue program execution at the point of an arithmetic trap. Software can force a trap to be continued more easily without the need for complicated fixup code. This is accomplished by following a set of code generation restrictions in the code that could cause arithmetic traps

which are to be completed by a software trap handler (see *Common Architecture, Chapter 4*), including specifying the /S software completion modifier in each such instruction.

The AND of all the software completion modifiers for trapping instructions is provided to the arithmetic trap handler in the exception summary SWC bit. If the SWC is set, a trap handler may find the trigger instruction by scanning backward from the trap PC until each register in the register write mask has been an instruction destination. The trigger instruction is the first instruction in the I-stream order to get a trap within a trap shadow. (See *Common Architecture, Chapter 4* for a definition of trap shadow.) If the SWC bit is clear, no fixup is possible.

- A synchronous trap occurs at the completion of the operation that caused the exception. No instructions can be issued between the completion of the operation that caused the exception and the trap.

5.1.2 Interrupts

The processor arbitrates interrupt requests. When the interrupt priority level (IPL) of an outstanding interrupt is greater than the current IPL, the processor raises IPL to the level of the interrupt and dispatches to entInt, the interrupt entry to the OS. Interrupts are serviced in kernel mode on the kernel stack. Interrupts can come from one of four sources: I/O devices, the clock, performance counters, or machine checks.

5.2 Processor Status

The processor status (PS) is a four-bit register that contains the current mode (PS<mode>) in bit <3> and a three-bit interrupt priority level (PS<IPL>) in bits <2..0>. The PS<mode> bit is zero for kernel mode and one for user mode. The PS<IPL> bits are always zero if the mode is user and can be 0 to 7 if the mode is kernel. The PS is changed when an interrupt or exception is initiated and by the rti, retsys, and swpipl instructions.

The uses of the PS values are shown in Table 5–1.

Table 5–1: Processor Status Summary

PS<mode>	PS<IPL>	Mode	Use
1	0	User	User software
0	0	Kernel	System software
0	1	Kernel	System software
0	2	Kernel	System software
0	3	Kernel	Low priority device interrupts
0	4	Kernel	High priority device interrupts

Table 5–1 (Cont.): Processor Status Summary

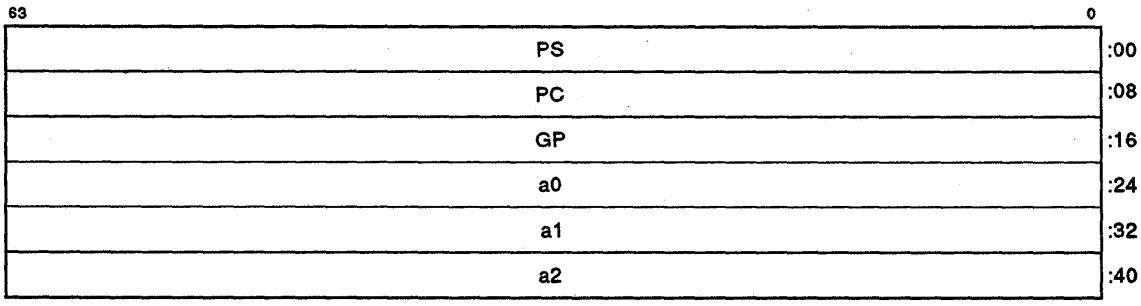
PS<mode>	PS<IPL>	Mode	Use
0	5	Kernel	Clock, and interprocessor interrupts
0	6	Kernel	Real time devices
0	7	Kernel	Machine checks

5.3 Stack Frames

There are two types of system entries—those for the callsys instruction and those for exceptions and interrupts. Both types use the same stack frame layout, as shown in Figure 5–1. The stack frame contains space for the PC, the PS, the saved GP, and the saved registers a0, a1, a2. On entry, the SP points to the saved PS.

The callsys entry saves the PC, the PS, and the GP. The exception and interrupt entries save the PC, the PS, the GP, and also save the registers a0..a2.

Figure 5–1: Stack Frame Layout



5.4 System Entry Addresses

All system entries are in kernel mode. The interrupt priority PS bits (PS<IPL>) are set as shown in the following table. The system entry point address is set by the CALL_PAL wrent instruction, as described in Section 2.2.10.

Table 5–2: Entry Point Address Registers

Entry Point	Value in a0	Value in a1	value in a2	PS<IPL>
entArith	Exception summary	Register mask	UNPREDICT-ABLE	Unchanged
entIF	Fault Type code	UNPREDICT-ABLE	UNPREDICT-ABLE	Unchanged

Table 5–2 (Cont.): Entry Point Address Registers

Entry Point	Value in a0	Value in a1	value in a2	PS<IPL>
entInt	Interrupt type	Vector	UNPREDICT-ABLE	Priority of interrupt
entMM	VA	MMCSR	Cause	Unchanged
entSys	p0	p1	p2	Unchanged
entUna	VA	Opcode	Src/Dst	Unchanged

5.4.1 System Entry Arithmetic Trap (entArith)

The arithmetic trap entry, entArith, is called when an arithmetic trap occurs. On entry, a0 contains the exception summary register and a1 contains the exception register write mask. Section 5.4.1.1 describes the exception summary register and Section 5.4.1.2 describes the register write mask.

5.4.1.1 Exception Summary Register

The exception summary register, shown in Figure 5–2 and described in Table 5–3, records the various types of arithmetic exceptions that can occur together. Those types of exceptions are listed and described in Table 5–3.

Figure 5–2: Exception Summary Register

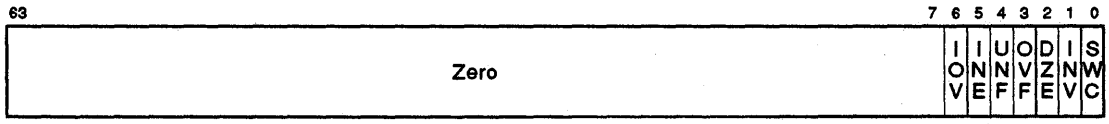


Table 5–3: Exception Summary Register Bit Definitions

Bit	Description
0	Software completion (SWC) Is set when all of the other arithmetic exception bits were set by floating-operate instructions with the /S software completion trap modifier set. See *Common Architecture, Chapter 4* for rules about setting the /S modifier in code that may cause an arithmetic trap, and Section 5.1.1 for rules about using the SWC bit in a trap handler.

Table 5–3 (Cont.): Exception Summary Register Bit Definitions

Bit	Description
1	Invalid operation (INV) An attempt was made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values were illegal. An INV trap is reported for most floating-point operate instructions with an input operand that is an IEEE NaN, IEEE infinity, or IEEE denormal. Floating invalid operation traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.
2	Division by zero (DZE) An attempt was made to perform a floating divide operation with a divisor of zero. A DZE trap is reported when a finite number is divided by zero. Floating divide by zero traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.
3	Overflow (OVF) A floating arithmetic or conversion operation overflowed the destination exponent. An OVF trap is reported when the destination's largest finite number is exceeded in magnitude by the rounded true result. Floating overflow traps are always enabled. If this trap occurs, the result register is written with an UNPREDICTABLE value.
4	Underflow (UNF) A floating arithmetic or conversion operation underflowed the destination exponent. An UNF trap is reported when the destination's smallest finite number exceeds in magnitude the non-zero rounded true result. Floating underflow trap enable can be specified in each floating-point operate instruction. If underflow occurs, the result register is written with a true zero.
5	Inexact result (INE) A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result. An INE trap is reported if the rounded result of an IEEE operation is not exact. Inexact result trap enable can be specified in each IEEE floating-point operate instruction. The rounded result value is stored in all cases.
6	Integer overflow (IOV) An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision. An IOV trap is reported for any integer operation whose true result exceeds the destination register size. Integer overflow trap enable can be specified in each arithmetic integer operate instruction and each floating-point convert-to-integer instruction. If integer overflow occurs, the result register is written with the truncated true result.

5.4.1.2 Exception Register Write Mask

The exception register write mask parameter records all registers that were targets of instructions that set the bits in the exception summary register. There is a one-to-one correspondence between bits in the register write mask quadword and the register numbers. The quadword records, starting at bit 0 and proceeding right to left, which of the registers r0 through r31, then f0 through f31, received an exceptional result.

NOTE

For a sequence such as:

```
ADDF F1,F2,F3
MULF F4,F5,F3
```

if the add overflows and the multiply does not, the OVF bit is set in the exception summary, and the F3 bit is set in the register mask, even though the overflowed sum in F3 can be overwritten with an in-range product by the time the trap is taken. (This code violates the destination reuse rule for software completion. See *Common Architecture, Chapter 4* for the destination reuse rules.)

The PC value saved in the exception stack frame is the virtual address of the next instruction. This is defined as the virtual address of the first instruction not executed after the trap condition was recognized.

5.4.2 System Entry Instruction Fault (entIF)

The instruction fault entry is called for bpt, bugchk, gentrap, opDec, and for a FEN fault (floating-point instruction when the floating-point unit is disabled, FEN EQ 0). On entry, a0 contains a 0 for a bpt, a 1 for bugchk, a 2 for gentrap, a 3 for FEN fault, and a 4 for opDec. No additional data is passed in a1..a2. The saved PC at (SP+00) is the address of the instruction that caused the fault for FEN faults. The saved PC at (SP+00) is the address of the instruction after the instruction that caused the fault bpt, bugchk, gentrap, and opDec faults.

5.4.3 System Entry Hardware Interrupts (entInt)

The interrupt entry is called to service a hardware interrupt, or a machine check. Table 5–4 shows what is passed in a0..a2 and the PS<IPL> setting for various interrupts.

Table 5–4: System Entry Hardware Interrupts

Entry Type	Value in a0	Value in a1	value in a2	PS<IPL>
Interprocessor interrupt	0	UNPREDICT-ABLE	UNPREDICT-ABLE	5

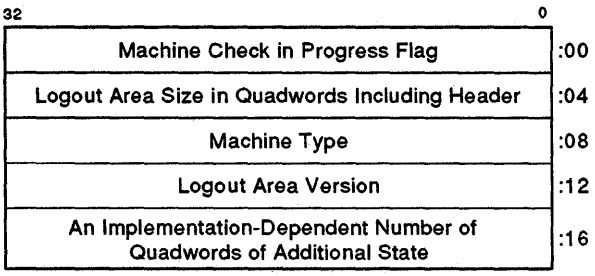
Table 5–4 (Cont.): System Entry Hardware Interrupts

Entry Type	Value in a0	Value in a1	value in a2	PS<IPL>
Clock	1	UNPREDICT-ABLE	UNPREDICT-ABLE	5
Machine check	2	Interrupt vector	Pointer to Logout Area	7
I/O device interrupt	3	Interrupt vector	UNPREDICT-ABLE	Level of device
Performance counter	4	Interrupt vector	UNPREDICT-ABLE	6

On entry to the hardware interrupt routine, the IPL has been set to the level of the interrupt. For hardware interrupts, register a1 contains a platform-specific interrupt vector. That platform-specific interrupt vector is typically the same value as the SCB offset value that would be returned if the platform was running OpenVMS PALcode.

For a machine check, a2 contains kseg address of the logout area. The first 4 longwords of the logout area are implementation-independent. The rest of the logout area is system specific. The first longword of the logout area is a machine check in progress flag. If the flag is non zero when a machine check is being initiated, a double machine check halt is initiated instead. The machine check handler needs to clear the machine check in progress flag when it can handle a new machine check. Figure 5–3 describes the logout area.

Figure 5–3: Logout Area



5.4.4 System Entry MM Fault (entMM)

The memory-management fault entry is called when a memory management exception occurs. On entry, a0 contains the faulting virtual address and a1 contains the MMCSR (See Section 3.10). On entry, a2 is set to a minus one (−1) for an instruction fetch fault, to a plus one (+1) for a fault caused by a store instruction, or to a 0 for a fault caused by a load instruction.

5.4.5 System Entry Call System (entSys)

The system call entry is called when a callsys instruction is executed in user mode. On entry, only registers (t8..t11) have been modified. The PC+4 of the callsys instruction, the user global pointer, and the current PS are saved on the kernel stack. Additional space for a0..a2 is allocated. After completion of the system service routine, the kernel code executes a CALL_PAL retsys instruction.

5.4.6 System Entry Unaligned Access (entUna)

The unaligned access entry is called when a load or store access is not aligned. On entry, a0 contains the faulting virtual address, a1 contains the zero extended six-bit opcode (bits <31:26>) of the faulting instruction, and a2 contains the zero extended data source or destination register number (bits<25:21> of the faulting instruction)

5.5 PALcode Support

5.5.1 Stack Writeability and Alignment

PALcode only accesses the kernel stack. Any PALcode accesses to the kernel stack that would produce a memory-management fault will result in a kernel-stack-not-valid halt. The stack pointer must always point to a quadword-aligned address. If the kernel stack is not quadword aligned on a PALcode access, a kernel-stack-not-valid halt is initiated.

5.6 \Revision History

Revision 1.0, May 12, 1992

- First review distribution

\

Platforms (IV)

This part describes an architected platform implementation and contains the following chapters:

- Chapter 1, Console Subsystem Overview and Operator Interface (IV)
- Chapter 2, Console Interface to Operating System Software (IV)
- Chapter 3, System Bootstrapping (IV)

Contents

Platforms (IV)

Chapter 1 Console Subsystem Overview and Operator Interface (IV)

Chapter 2 Console Interface to Operating System Software (IV)

Chapter 3 System Bootstrapping (IV)

Figures

Tables

Chapter 1
Console Subsystem Overview and Operator Interface (IV)

On an Alpha system, underlying control of the system platform hardware is provided by a "console" [1] . The console:

1. Initializes, tests, and prepares the system platform hardware for Alpha system software.
2. Bootstraps (loads into memory and starts the execution of) system software.
3. Controls and monitors the state and state transitions of each processor in a multiprocessor system.
4. Provides services to system software which 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. The actual mechanisms of these interactions are obviously specific to the platform hardware, however the net effects are common to all systems. Chapter 3 describes these functions.

The console interacts with system software once control of the system platform hardware has been transferred to that software. Chapter 2 discusses the basic functions of a console and its interaction with Alpha system software.

The console interacts with the console operator through a virtual display device or "console terminal". The console operator may be a human or a management application. The console terminal forms the interface between the console and a console presentation layer. The functions of that presentation layer and the display formats are described in Section 1.3.

In an Alpha multiprocessor system, there is one primary processor and one or more secondary processors. The primary is the processor that:

1. Can legally refer to the console I/O devices,
2. Can legally send characters to the console terminal,
3. Can legally receive characters from the console terminal,
4. Has direct access to a BB_WATCH on the system

[1] A term shrouded in the antiquity of computing. So named because this mechanism was first realized as a desklike panel of switches and blinking lights.

5. Is named in response to an inquiry as to which processor is primary.

All other processors in the system are secondary processors.

1.1 Console Implementations

The actual implementation of an Alpha console varies from system to system. Regardless of implementation, the console on each system provides the functionality described in this chapter and in Chapters 2 and 3. The console may be implemented as:

- "Embedded" or co-resident in the hardware platform complex which contains the processors.
- "Detached" or resident on a separate and distinct hardware platform.
- Any hybrid of the above.

The distinction is somewhat arbitrary. A detached console may have cooperating special code which executes on one of the processors; an embedded console may have a cooperating management application which executes on a remote machine.

Regardless of the actual implementation, each console must provide:

1. A virtual display device, the default "console terminal".

   This device allows the console operator to issue commands and receive displays. In the absence of hardware errors and with the proper console-lock setting, the default console terminal device provides reliable communication with the rest of the console.

2. Reliable access to console functionality by system software and the console operator.

   All console functionality must appear to be resident within the console at all times. All console functions must be accessible in a timely manner, without prior notification, and with sufficient reliability.

3. Secure communications with system software and the console operator.

   All console communication paths must be able to be made secure by either physical measures or encryption methods.

4. A mechanism by which the console can gain control of a processor executing system software.

   This mechanism must preserve the execution state of system software; it must be possible for the console to gain control of the processor, and subsequently continue system software execution successfully.

5. A mechanical mechanism which locks the console.

   The console lock may be a keyswitch, jumper, or any other implementation-specific mechanism; see Section 1.2. The lock is either "locked" or "unlocked".

1.1.1 Console Implementation Registry

This chapter, and Chapters 2 and 3 specify required console functions. Some of these functions have attributes which may vary with console implementation; consoles may also extend beyond the required functions. Console functions or attributes which may vary with implementation are:

1. Supported CTBs
2. Supported environment variables
3. Environment variable value formats, such as BOOT_DEV or BOOT_OSFLAGS
4. Configuration Data Block format
5. Supported callback routines
6. Supported bootstrap media
7. Implementation-specific HALT codes or messages

Functions implemented by current consoles are summarized in *Appendix E.* \Also see that appendix for information on how to register a function.\

1.2 Console Lock Mechanisms

TBD in a subsequent ECO.

1.3 Console Presentation Layer

The console presentation layer is TBD in a subsequent ECO. This text assumes the following command syntax:

- BOOT (bootstrap the system)
- CONTINUE (continue execution)
- START -CPU (start a given secondary)
- INITIALIZE (initialize system)
- INITIALIZE -CPU (initialize a given processor)
- HALT -CPU (force a given processor into console I/O mode)
- HALT -CRASH (cause a given processor to initiate a crash)

1.4 Messages

The console generates a binary message code to the console presentation layer to signal messages, such as audit trail or error messages. The console presentation layer interprets the binary code into something meaningful to the console operator. Table 1–1 summarizes the binary message codes, symbol names, and the expected translation into English.

Table 1–1: Console Error Messages

$Code_{16}$	Symbol	English Interpretation
1	AUDIT_BOOT_STARTS	Audit trail of booting begins
2	AUDIT_BSTRAP_GOOD	Bootstrap checksum matches
3	AUDIT_BSTRAP_ACCESSIBLE	Bootstrap image accessible
4	AUDIT_CHECKSUM_GOOD	Boot block checksum matches
5	AUDIT_LOAD_BEGINS	Loading of bootstrap begins
6	AUDIT_LOAD_DONE	Loading of bootstrap done
7	AUDIT_TAPE_ANSI	Verified as ANSI tape
8	AUDIT_FILE_FOUND<filename>	Found <filename>
9	AUDIT_TAPE_BBLOCK	Verified as bootblocked tape
A	AUDIT_BOOT_TYPE<string>	Bootstrap type <string>
B	AUDIT_BOOT_REQ<filename>	Requesting bootstrap<filename>
C	AUDIT_BSERVER_FOUND	Remote server located
D	AUDIT_BSTRAP_ABORT	Bootstrap load abort
E-3F	reserved	
40	?PALREQ?	PALcode load request
41	?STARTREQ?	Secondary start request
42-7F	reserved	
80	ERROR_BOOT_ABORT	Unable Boot
81	ERROR_PROC_INIT	Unable to Initialize Processor
82-FFF	reserved	
other	console	implementation-specific

1.5 Implementation Considerations

1.5.1 Console Implementations

\ This chapter and Chapters 2 and 3 attempt to standardize across all console implementations, the dissimilar options, functions, and features, that were not mentioned by DEC STD 032. The lack of standardization for VAX systems presented VAX software and Digital Field Service with too many different interfaces. \

The goal of the Alpha console architecture is to promote a consistent interface across all Alpha systems. Some console functionality is inherently implementation-specific and cannot be required of all Alpha systems; some may be applicable to more than

one Alpha system. To prevent the proliferation of interfaces and achieve commonality of function whenever possible, the Alpha console architecture requires that:

1. Any console function which is visible to system software which is not specified by these chapters must be registered with the Alpha architecture group.
2. Any console function which is visible to an on-site or remote console operator (including Field Service engineers) which is not specified by these chapters must be registered with the Alpha architecture group.
3. Whenever possible, implementations must use previously registered functions rather than inventing new variations.

Console functions intended for use solely by development engineering or expert-level repair and diagnosis are excluded from the above. See *Appendix E* for registry information.

1.5.2 Security

The means by which the console achieves a secure communications path with system software and with the console operator is implementation-specific. Embedded consoles inherently have the capability of secure communications with system software. Detached consoles can achieve this security by residing in the same room as the Alpha system and communicating with it over a private connection. Detached consoles can also achieve security by using an encrypted protocol over a shared connection. This latter method allows a workstation over a network to function as the console.

1.5.3 Internationalization

Wherever possible, console implementations should support the goals of internationalization:

1. Each message has a binary message code. The console presentation layer interprets the code into a meaningful message display of the appropriate language and characters.
2. Consoles should avoid explicitly interpreting character set encoding (such as ISO–LATIN–1). Character strings are to be viewed as simple byte strings. Thus, the GETC console callback routine supports from one to four byte character encodings depending on the currently selected language and character set; the PUTS routine outputs only a byte stream.
3. ASCII strings are used in certain fields of the HWRPB and certain interprocessor communications due to DEC Standard 12 and to present a common interface to system software.
4. The currently selected character-set encoding and language to be used for the console terminal are defined by the CHAR_SET and LANGUAGE environment variables.
5. The end of a character string passed between the console and the operating system as an argument to a console callback routine is determined by passing its length.

6. Console callback routines should be written to be independent from character-set encoding and language. At a minimum, every implementation must support ISO–LATIN–1 character-set encodings. The supported character-set encodings is determined by platform product requirements.
7. The console presentation layer is independent of the required console functionality interface.

1.5.4 ISO–LATIN–1 Support

Implementations supporting the ISO-LATIN-1 character-set encoding must have the following properties:

1. The GETC console callback routine returns a one byte character; see Section 2.3.4.
2. The PROCESS_KEYCODE console callback routine returns a one byte character; see Section 2.3.4
3. English console presentation layers are strongly encouraged to use the actual values as defined in Table 2–5, rather than inventing aliases.

1.6 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Reorganized according to SRM Rev 5 requirements
2. Converted to SDML
3. Replace previous Console Chapter with Console ECO #15
4. Includes 3 chapters and two appendices, renumber I/O Chapter
5. Material substantially changed or rearranged

\

Chapter 2

Console Interface to Operating System Software (IV)

This chapter describes the interactions between the console subsystem and system software. These services depend on state which is shared between the console and system software. That shared state is contained in the "Hardware Restart Parameter Block" (HWRPB) and a number of "environment variables". The HWRPB is a data structure which is directly accessed by both the console and system software; the environment variables are indirectly accessed by system software. Section 2.1 describes the HWRPB; Section 2.2 describes the environment variables. The service, or "callback", routines provided by the console to system software are given in Section 2.3. Communication between the console and system software is described in Section 2.4. Functions implemented by registered consoles are summarized in *Appendix E*. Various implementation considerations are given in Section 2.5.

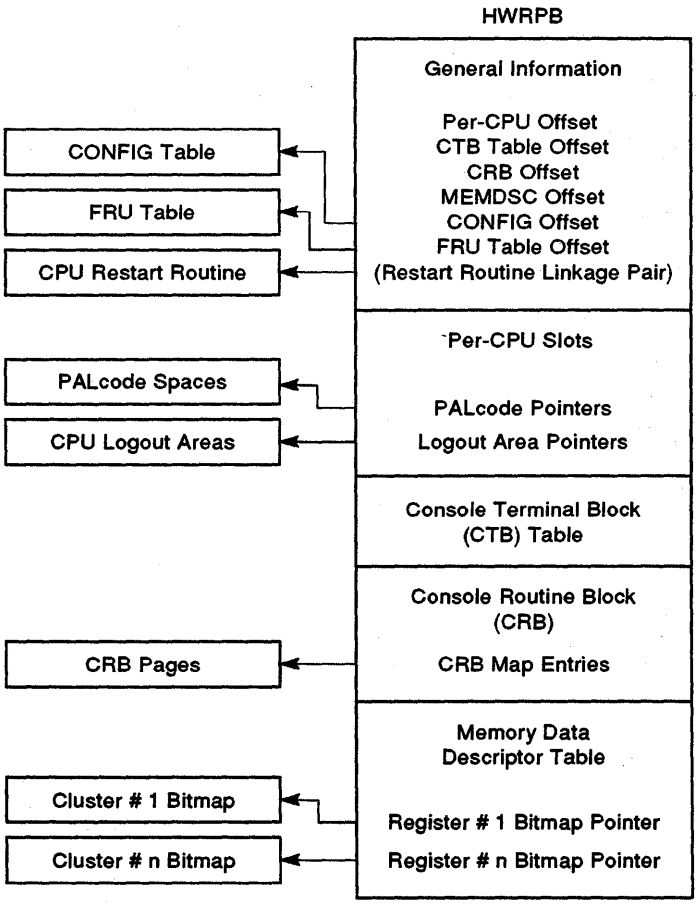
2.1 Hardware Restart Parameter Block (HWRPB)

The Hardware Restart Parameter Block (HWRPB) is a page-aligned data structure shared between the console and system software. The HWRPB is a critical resource during bootstraps, powerfail recoveries, and other restart situations. The fields of the HWRPB are shown in Figure 2–1 and described in Table 2–1.

The console creates the HWRPB and the required per-CPU, CTB, CRB, and MEMDSC offset blocks as a physically contiguous structure during console initialization. Fields within the HWRPB and the required offset blocks are updated by the console and system software during and after system bootstrapping. The console must be able to locate the HWRPB and the required offset blocks at all times. Neither the console nor system software may move the HWRPB or the required offset blocks to different physical memory locations; subsequent operation of the system is UNDEFINED if such an attempt is made.

The HWRPB and the required offset blocks must comprise a virtually contiguous structure at all times. Prior to transferring control to system software, the console maps the HWRPB and the required offset blocks into contiguous addresses beginning at virtual address 0000 0000 1000 0000_{16}. in the initial bootstrap address space. If system software subsequently changes this virtual mapping, any new mapping must preserve the relative offsets of all fields and blocks; all physically contiguous pages must remain virtually contiguous. Note that some of the data structures located by HWRPB fields need not be contiguous with the HWRPB. Those structures which may be discontiguous are the optional CONFIG Block, the optional FRU Table, the PALcode space(s), the logout area(s), the CRB pages, and the memory bitmaps located by the MEMDSC Table.

Figure 2–1: HWRPB Overview



All offset blocks must be at least quadword aligned. The starting address of an offset block is determined by adding the contents of the HWRPB offset field to the starting address of the HWRPB. For example, the starting address of the MEMDSC block is given by:

```
MEMDSC Address = HWRPB address + MEMDSC OFFSET
               = HWRPB address + (HWRPB[200])
```

The total size of the HWRPB and the required offset blocks is on the order of 8KB to 16KB. The size is contained in the HWRPB_SIZE field at HWRPB[24]. The required offset blocks may be offset from the HWRPB in any order; the HWRPB offset fields must not be used to infer the size of the HWRPB nor any offset block.

Figure 2–2: Hardware Restart Parameter Block Structure

63 — 0	
Physical Address of the HWRPB	:HWRPB
"HWRPB"	:+08
HWRPB Revision	:+16
HWRPB Size	:+24
Primary CPU ID	:+32
Page Size (Bytes)	:+40
Number of PA Bits	:+48
Maximum Valid ASN	:+56
System Serial Number (SSN)	:+64
System Type	:+80
System Variation	:+88
System Revision	:+96
Interval Clock Interrupt Frequency	:+104
Cycle Counter Frequency	:+112
Virtual Page Table Base	:+120
Reserved for Architecture Use	:+128
Offset to Translation Buffer Hint Block	:+136
Number of Processor Slots	:+144
Per-CPU Slot Size	:+152
Offset to Per-CPU Slots	:+160
Number of CTBs	:+168
CTB Size	:+176
Offset to Console Terminal Block Table	:+184
Offset to Console Callback Routine Block	:+192
Offset to Memory Data Descriptor Table	:+200
Offset to Configuration Data Block (If Present)	:+208
Offset to FRU Table (If Present)	:+216
Virtual Address of Terminal Save State Routine	:+224
Procedure Value of Terminal Save State Routine	:+232
Virtual Address of Terminal Restore State Routine	:+240
Procedure Value of Terminal Restore State Routine	:+248

Figure 2–2 (continued on next page)

Figure 2–2 (Cont.): Hardware Restart Parameter Block Structure

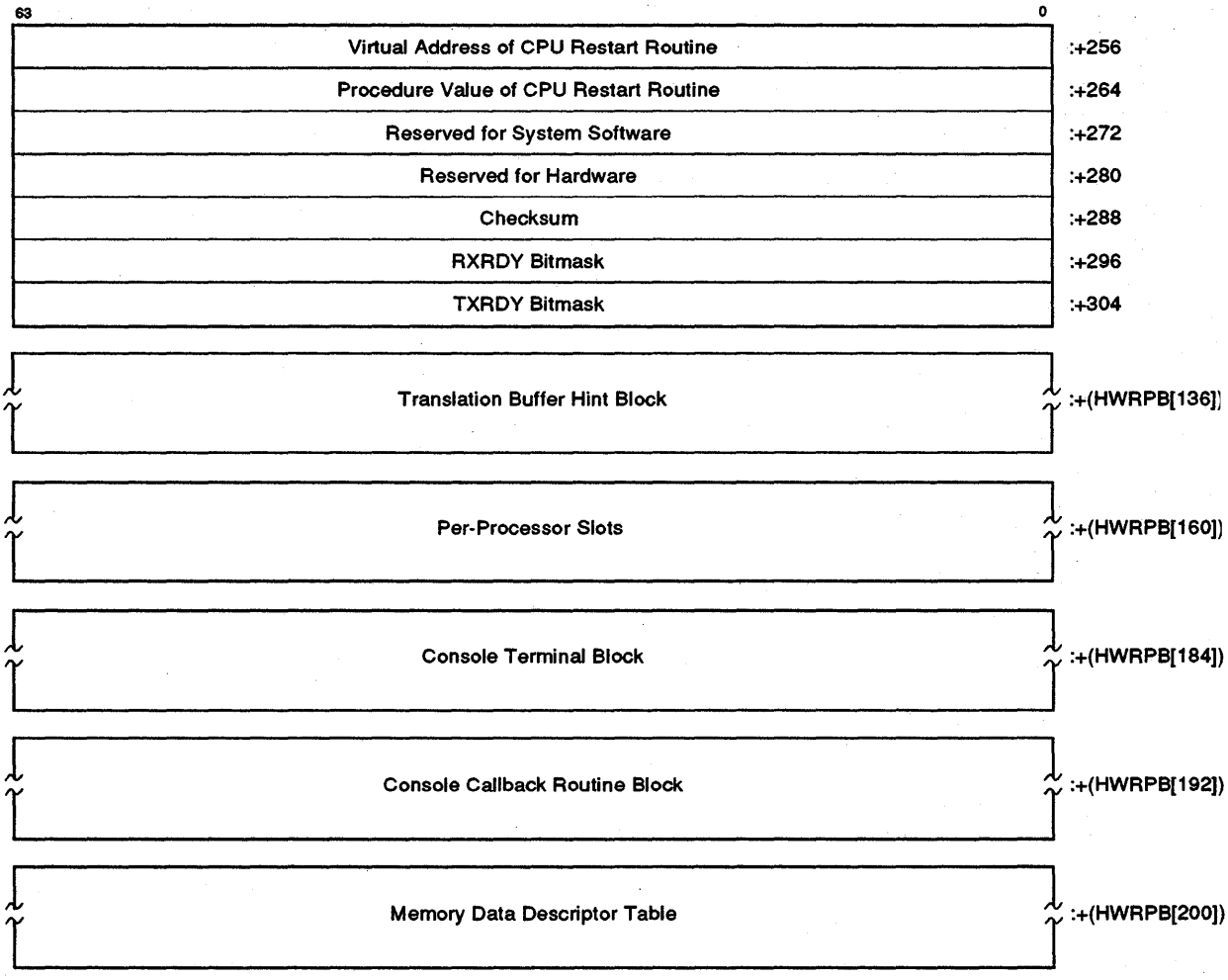


Table 2–1: HWRPB Fields

Offset	Description
HWRPB	HWRPB PA[1] Starting physical address of the HWRPB field. This field is used by the console to validate the HWRPB.
+08	HWRPB VALIDATION[1] Quadword containing "HWRPB<0><0><0>" ($0000\ 0042\ 5052\ 5748_{16}$). This field is used by the console to validate the HWRPB.

[1]Initialized by the console at cold system bootstrap only. Preserved unchanged by the console at all warm system bootstraps.

Table 2–1 (Cont.): HWRPB Fields

Offset	Description
+16	HWRPB REVISION[1] Format of the HWRPB. See Section 2.1.1. Assigned values are referenced to the revision level of this chapter:

Version	Interpretation
0	Reserved
1	Revision 1.1–2.1 \ADU only\
2	Revision 3.0
3	Revision 3.3 \ECO #30\
other	Reserved for future use

+24	HWRPB SIZE[1] Size in bytes of the HWRPB and required physically contiguous per-CPU, CTB, CRB, and MEMDSC offset blocks. Unsigned field.
+32	PRIMARY CPU ID[1,4] WHAMI of the primary processor. System software modifies this field only at primary switch; see Section 3.4.6. Unsigned field.
+40	PAGE SIZE[1] Number of bytes within a page for this Alpha processor implementation. Unsigned field.
+48	PA SIZE[1] Size of the physical address space in bits for this Alpha processor implementation. PA SIZE must be 48 bits or less; see *OpenVMS Section, Chapter 3* and *Common Architecture, Chapter 5*. Unsigned field.
+56	MAX VALID ASN[1] Maximum ASN value allowed by this Alpha processor implementation. Unsigned field.
+64	SYSTEM SERIAL NUMBER[1] Full DEC STD 12 serial number for this Alpha System. This octaword field contains a 10 character ASCII serial number determined at the time of manufacture; see DEC STD 12 for format information.
+80	SYSTEM TYPE[1] Family or system hardware platform. Assigned values are summarized in *Appendix D*; see Section 2.1.1. Unsigned field.

[1]Initialized by the console at cold system bootstrap only. Preserved unchanged by the console at all warm system bootstraps.

[4]May be modifed by system software.

Table 2–1 (Cont.): HWRPB Fields

Offset	Description
+88	SYSTEM VARIATION[1,4] Subtype variation of the system. This may include whether the system has optional features such as multiprocessor support or special power supply conditioning. Assigned values are summarized in *Appendix D*; see Section 2.1.1.
+96	SYSTEM REVISION CODE[1,4] DEC STD 12 revision field for this Alpha system. Four ASCII characters. May be modified by system software or application software.
+104	INTERVAL CLOCK INTERRUPT FREQUENCY[1] Number of interval clock interrupts per second (scaled by 4096) in this Alpha system. Interrupts occur only if enabled; see *OpenVMS Section, Chapter 6*. Unsigned field.
+112	CYCLE COUNTER FREQUENCY[1] Number of SCC and PCC updates per second in this Alpha system. See the RPCC and PAL RSCC instructions. Unsigned field.
+120	VIRTUAL PAGE TABLE BASE[2,4] Virtual address of the base of the entire three-level page table structure; see *OpenVMS Section, Chapter 6*. The console sets this field to the virtual address of the L1PTE within bootstrap address space at system bootstraps and restores VPTB IPR with this value at all processor restarts. System software is responsible for updating this field whenever the VPTB IPR is modified. See Sections 3.3.1.3, 3.3.3.5, and 3.4.2.
+128	Reserved Reserved for architecture use; SBZ.
+136	TB HINT OFFSET[1] Unsigned offset to the starting address of the Translation Buffer Hit Block (TBB). See Section 2.1.2.
+144	NUMBER OF PER-CPU SLOTS[1] Number of per-CPU slots present. See Section 2.1.3 for the per-CPU slot format. Unsigned field.
+152	PER-CPU SLOT SIZE[1] Size in bytes of each per-CPU slot rounded up to the next integer multiple of 128. See Section 2.1.3. Unsigned field.
+160	CPU SLOT OFFSET[1] Unsigned offset to the first per-CPU slot in the HWRPB. See Section 2.1.3.

[1]Initialized by the console at cold system bootstrap only. Preserved unchanged by the console at all warm system bootstraps.

[2]Initialized by the console at all system bootstraps (cold or warm).

[4]May be modifed by system software.

Table 2–1 (Cont.): HWRPB Fields

Offset	Description
+168	NUMBER OF CTB[1] Number of Console Terminal Blocks (CTBs) contained in the CTB Table. See Section 2.3.8.2. Unsigned field.
+176	CTB SIZE[1] Size in bytes of the largest Console Terminal Block (CTB) contained in the CTB Table. See Section 2.3.8.2. Unsigned field.
+184	CTB OFFSET[1] Unsigned offset to the starting address of the Console Terminal Block (CTB) Table. See Section 2.3.8.2.
+192	CRB OFFSET[1] Unsigned offset to the starting address of the Console Callback Routine Block (CRB). See Section 2.3.8.1.
+200	MEMDSC OFFSET[1] Unsigned offset to the starting address of the Memory Data Descriptor (MEMDSC) Table. See Section 3.3.1.1.
+208	CONFIG OFFSET[1] Unsigned offset to the starting address of the Configuration Data Table (CONFIG). If zero, no CONFIG Table exists. See Section 2.1.4.
+216	FRU TABLE OFFSET[1] Unsigned offset to the starting address of the Field Replaceable Unit (FRU) Table. If zero, no FRU Table exists. See Section 2.1.5.
+224	SAVE_TERM RTN VA[2,4] Starting virtual address of a routine which saves console terminal state. This routine is optionally provided by system software. See Section 3.4.7. Set to zero by the console at system bootstraps.
+232	SAVE_TERM VALUE[2,4] Procedure value of the SAVE_TERM routine optionally provided by system software. The console copies this value into R27 before invoking the routine; see Section 3.4.7. Set to zero by the console at system bootstraps.
+240	RESTORE_TERM RTN VA[2,4] Starting virtual address of a routine which restores console terminal state. This routine is optionally provided by system software. See Section 3.4.7. Set to zero by the console at system bootstraps.

[1]Initialized by the console at cold system bootstrap only. Preserved unchanged by the console at all warm system bootstraps.

[2]Initialized by the console at all system bootstraps (cold or warm).

[4]May be modifed by system software.

Table 2–1 (Cont.): HWRPB Fields

Offset	Description
+248	RESTORE_TERM VALUE[2,4] Procedure value of the RESTORE_TERM routine optionally provided by system software. The console copies this value into R27 before invoking the routine; see Section 3.4.7. Set to zero by the console at system bootstraps.
+256	RESTART RTN VA[2,4] Starting virtual address of a CPU restart routine provided by system software. The console restarts system software by transferring control to this routine. See Section 3.4. Set to zero by the console at system bootstraps.
+264	RESTART VALUE[2,4] Procedure value of the CPU restart routine provided by system software. During the restart process, the console copies this value into R27 before transferring control to the CPU restart routine. See Section 3.4. Set to zero by the console at system bootstraps.
+272	RESERVED FOR SYSTEM SOFTWARE[2,4] Reserved for use by system software. Set to zero by the console at system bootstraps.
+280	RESERVED FOR HARDWARE[1] Reserved for use by hardware.
+288	HWRPB CHECKSUM[2,4] Checksum of all the quadwords of the HWRPB from offset [00] to [118] inclusive. Computed as a 64-bit, 2's complement sum ignoring overflows. Used to validate the HWRPB during warm bootstraps, restarts, and secondary starts. Set by console initialization; recomputed and updated whenever a HWRPB field with offset [00] to [118] inclusive is modified by the console or system software.
+296	RXRDY BITMASK[2,4] Secondary receive bitmask for interprocessor console communications. When transmitting a command to a secondary, the primary processor sets the RXRDY bit which corresponds to the CPU ID of the secondary. The number of active bits in this field is determined by the number of per-CPU slots in HWRPB[144]. See Section 2.4. All bits are initialized as clear.
+304	TXRDY BITMASK[2,4] Secondary transmit bitmask for interprocessor console communications. When transmitting a message to the primary, the secondary processor sets the TXRDY bit which corresponds to its CPU ID and requests an interprocessor interrupt to the primary. The number of active bits in this field is determined by the number of per-CPU slots in HWRPB[144]. See Section 2.4. All bits are initialized as clear.

[1]Initialized by the console at cold system bootstrap only. Preserved unchanged by the console at all warm system bootstraps.

[2]Initialized by the console at all system bootstraps (cold or warm).

[4]May be modifed by system software.

Table 2–1 (Cont.): HWRPB Fields

Offset	Description
+(HWRPB[136])	TB HINT BLOCK[2,4] Quadword-aligned block that describes the characteristics of the translation buffer (TB) granularity hints. See Section 2.1.2.
+(HWRPB[160])	Per-CPU SLOTS[2,4] 128 Byte-aligned slots which describe each processor in the system. See Section 2.1.3.
+(HWRPB[184])	CTB TABLE[1] Quadword-aligned Console Terminal Block Table. Set at console initialization; modified by console terminal callbacks. See Section 2.3.8.2.
+(HWRPB[192])	CONSOLE CALLBACK ROUTINE BLOCK[2,4] Quadword-aligned block that describes the location and mapping of the console callback routines. Set at system bootstrap; modified by console FIXUP callback. See Section 2.3.8.1.
+(HWRPB[200])	MEMDSC[1,4] Quadword-aligned Memory Data Descriptor Table. Set at console initialization; preserved across warm bootstraps. See Section 3.3.1.1.

[1]Initialized by the console at cold system bootstrap only. Preserved unchanged by the console at all warm system bootstraps.
[2]Initialized by the console at all system bootstraps (cold or warm).
[4]May be modifed by system software.

2.1.1 Revision, Type, and Variation Fields

The HWRPB contains several revision, type, and variation fields which describe the Alpha system platform hardware and PALcode. System software uses these fields to identify hardware-dependent support code which must be loaded or enabled. These fields are examined early in operating system bootstrap; if one of the fields contains a value which is unrecognized or incompatible with the operating system, the bootstrap attempt fails. Diagnostic software uses these fields to guide field installation and upgrade procedures and for material and parts control.

In multiprocessor systems, the processor type and PALcode revisions need not be identical for all processors. System software uses these fields to determine if multiprocessor operation is viable. This evaluation may be performed by running primary, the starting secondary, or a combination of both. For example, see Section 3.3.3.3. The fields include:

1. HWRPB Revision - HWRPB[16]

   This field identifies the format of the HWRPB. Since the HWRPB is shared between the console and system software, both must agree on the field offsets, formats, and interpretations.

2. System Type and System Variation - HWRPB[80] and HWRPB[88]

These fields identify the Alpha system platform. System software infers attributes such as physical address offsets and I/O device locations from the system type.

3. System Revision - HWRPB[96]

   This field identifies the system platform hardware revision.

4. Processor Type and Processor Variation - SLOT[176] and SLOT[184]

   These per-CPU slot fields identify each Alpha processor and its capabilities. The Processor Type field contains two sub-fields. The major type sub-field identifies the processor implementation \ (such as EV–3 or EV–4) \ ; the minor type sub-field identifies any system-specific attributes (such as local memory or cache size)

5. Processor Revision - SLOT[192]

   This per-CPU slot field identifies the processor hardware revision.

6. PALcode Revision - SLOT[168]

   This field identifies the PALcode revision required and/or in use by the processor. System software uses the PALcode variation and PALcode compatibility sub-fields. The variation subfield indicates whether the PALcode image includes extensions or functional variations necessary to a given operating system or application.

PROGRAMMING NOTE

For example, a PALcode variation may contain a different TB fill routine. System software uses the compatibility subfield to ensure that all processors in a multiprocessor system are using compatible PALcode images.

\ PALcode revisions are specific to the system platform and processor major type. The filename of distributed PALcode images must contain sufficient information to distinguish the intended system platform and processor. \

2.1.2 Translation Buffer Hint Block

The Translation Buffer Hint Block (TBB) contains information on the characteristics of the instruction stream translation buffer (ITB) and data stream translation buffer (DTB) granularity hints (GH). All processors in a multiprocessor Alpha system must implement the same granularity hints.

The TBB consists of 8 quadwords, 4 for each of the translation buffers (ITB and DTB). The 4 quadwords contain 16 word fields; each word contains the number of entries in the translation buffer that implement a combination of granularity hints (including none).

Table 2–2: Granularity Hint Fields

$Offset_{16}$	Granularity Hint
0	None
2	1 page
4	8 pages
6	1 and 8 pages
8	64 pages
A	1 and 64 pages
C	8 and 64 pages
E	1, 8, and 64 pages
10	512 pages
12	1 and 512 pages
14	8, and 512 pages
16	1, 8 and 512 pages
18	64 and 512 pages
1A	1, 64, and 512 pages
1C	8, 64, and 512 pages
1E	1, 8, 64, and 512 pages

2.1.3 Per-CPU Slots in the HWRPB

Information on the state of a processor is contained in a "per-CPU slot" data structure for that processor. The per-CPU slots form a contiguous array indexed by CPU ID. The starting address of the first per-CPU slot is given by the offset HWRPB[160] relative to the starting address of the HWRPB. The number of per-CPU slots is given in HWRPB[144]. Each per-CPU slot must be 128B aligned to ensure natural alignment of the HWPCB at SLOT[0]. The slot size rounded up to the nearest multiple of 128 bytes, is given in HWRPB[152].

CPU IDs are determined in an implementation-specific manner. The only requirement is that they be in the range of zero to the maximum number of processors the particular platform supports minus one.

SOFTWARE NOTE

OpenVMS Alpha supports CPU IDs in the range 0–31 only.

Each per-CPU slot contains information necessary to bootstrap, start, restart or halt the processor. The format is shown Figure 2–3 and Table 2–3. The HWPCB specifies the context in which the loaded system software will execute; see *OpenVMS Section, Chapter 4* for more information.

The console must initialize the per-CPU slot for the primary processor prior to system bootstrap. The per-CPU slot fields for secondary processors are set by a combination of the console and system software. The console updates the halt information at error halts and prior to processor restarts.

Slots corresponding to nonexistent processors are zeroed. There may be more per-CPU slots than are necessary in any given Alpha system. A system implementation may reserve HWRPB space for processors which are not present at system bootstrap.

An Alpha system may support internally different, yet software compatible, PALcode for different processors in a multiprocessor implementation. Each per-CPU slot contains a PALcode memory descriptor which locates the PALcode used by that processor. See Section 3.3.1.2 for information on PALcode loading and initialization on the primary processor and Section 3.3.3.3 for information on PALcode loading and initialization on secondary processors.

The starting address of a per-CPU slot is calculated by:

```
Slot Address = {CPU ID * slot size}  + offset     + HWRPB base
             = {CPU ID * HWRPB[152]} + HWRPB[160] + #HWRPB
```

The address may be physical or virtual.

Figure 2–3: Per-CPU Slot in HWRPB

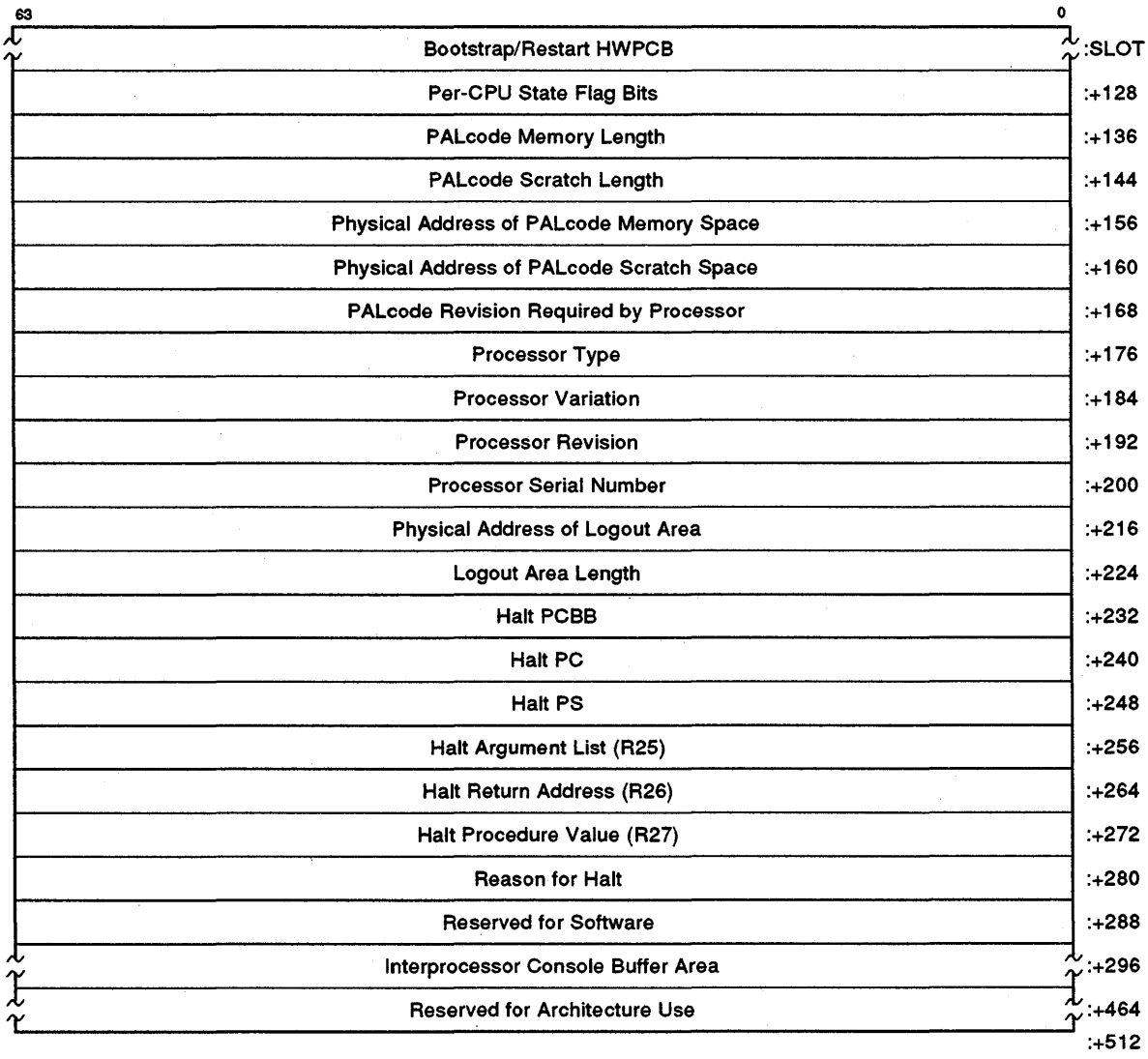


Table 2–3: Per-CPU Slot Fields

Offset	Description
SLOT	HWPCB[3,6] Hardware Privileged Context Block for this processor. See *OpenVMS Section, Chapter 4* for the structure of the HWPCB; see Table 3–6 for the contents as set by the console.
+128	STATE FLAGS[3,6] Current state of this processor. See Table 2–4 for the interpretation of each bit.
+136	PALCODE MEMORY SPACE LENGTH[1,2] Number of bytes required by this processor for PALcode memory. Unsigned field.
+144	PALCODE SCRATCH SPACE LENGTH[1,2] Number of bytes required by this processor for PALcode scratch space. Unsigned field.
+152	PA OF PALCODE MEMORY SPACE[1,6] Starting physical address of PALcode memory space for this processor. PALcode memory space must be page aligned. See Section 3.3.1.2 or Section 3.3.3.3.
+160	PA OF PALCODE SCRATCH SPACE[1,6] Starting physical address of PALcode scratch space for this processor. PALcode scratch space must be page aligned. See Section 3.3.1.2 or Section 3.3.3.3.

[1]Initialized by the console for primary at cold system bootstrap only. Preserved unchanged by the console at all other times.
[2]Initialized by the console for a secondary at cold system bootstrap only. Preserved unchanged by the console at all other times.
[3]Initialized by the console for the primary at all system bootstraps (cold or warm) and for a secondary prior to processor start.
[6]May by modified by system software for a secondary prior to processor start.

Table 2–3 (Cont.): Per-CPU Slot Fields

Offset	Description
+168	PALCODE REVISION[1,2] PALcode revision level for this processor.

Bits	Interpretation
<7:0>	PALcode minor revision (0–255)
<15:8>	PALcode major revision (0–255)
<23:16>	PALcode variation
	0 Reserved
	1 OpenVMS PALcode version
	2 DEC OSF/1 PALcode version
	3–127 Reserved for Digital
	128–255 Reserved for non-Digital
<31:24>	SBZ
<47:32>	PALcode compatibility (0–65535)
	0 Unknown
	1–65535 Compatibility revision
<63:48>	Maximum number of processors that can share this PALcode image

The major and minor PALcode revisions are set at console initialization; the remaining fields are set during PALcode loading and initialization. See Section 2.1.1 and Section 3.3.3.3.

+176 PROCESSOR TYPE[1,2]

Type of this processor.

Bits	Interpretation
<31:0>	Minor type
<63:32>	Major type

Assigned values are summarized in *Appendix D*; see Section 2.1.1.

+184 PROCESSOR VARIATION[1,2]

Variation or subtype of this processor. Assigned values are summarized in *Appendix D*; see Section 2.1.1.

[1]Initialized by the console for primary at cold system bootstrap only. Preserved unchanged by the console at all other times.
[2]Initialized by the console for a secondary at cold system bootstrap only. Preserved unchanged by the console at all other times.

Table 2–3 (Cont.): Per-CPU Slot Fields

Offset	Description
+192	PROCESSOR REVISION[1,2] Full DEC STD 12 revision field for this processor. This quadword field contains 4 ASCII characters. See Section 2.1.1.
+200	PROCESSOR SERIAL NUMBER[1,2] Full DEC STD serial number for this processor. This octaword field contains a 10 character ASCII serial number determined at the time of manufacture; see DEC STD 12 for format information.
+216	PA OF LOGOUT AREA[1,2] Starting physical address of PALcode logout area for this processor. Logout areas must be at least quadword aligned. See *OpenVMS Section, Chapter 6.*
+224	LOGOUT AREA LENGTH[1,2] Number of bytes in the PALcode logout area for this processor. See *OpenVMS Section, Chapter 6.*
+232	HALT PCBB[3,4] Value of the PCBB IPR when a processor halt condition is encountered by this processor. Initialized to the address of the HWPCB at offset [0] from this per-CPU slot at system bootstraps or secondary processor starts.
+240	HALT PC[3,4] Value of the PC when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.
+248	HALT PS[3,4] Value of the PS when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.
+256	HALT ARGUMENT LIST[3,4] Value of R25 (argument list) when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.
+264	HALT RETURN ADDRESS[3,4] Value of R26 (return address) when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.
+272	HALT PROCEDURE VALUE[3,4] Value of R27 (procedure value) when a processor halt condition is encountered by this processor. Zeroed at system bootstraps or secondary processor starts.

[1]Initialized by the console for primary at cold system bootstrap only. Preserved unchanged by the console at all other times.

[2]Initialized by the console for a secondary at cold system bootstrap only. Preserved unchanged by the console at all other times.

[3]Initialized by the console for the primary at all system bootstraps (cold or warm) and for a secondary prior to processor start.

[4]Set by the console at all processor halts.

Table 2–3 (Cont.): Per-CPU Slot Fields

Offset	Description
+280	REASON FOR HALT[3,4] Indicates why this processor was halted. Values include: $Code_{16}$ — Reason 0 — Bootstrap, processor start, or powerfail restart 1 — Console operator requested a system crash 2 — Processor halted due to kernel-stack not-valid halt 3 — Invalid SCBB 4 — Invalid PTBR 5 — Processor executed CALL_PAL HALT instruction in kernel mode 6 — Double error abort encountered 7-FFF — Reserved other — Implementation-specific See *OpenVMS Section, Chapter 6* for information on system exceptions associated with codes 2 through 6. Set to '0' at console initialization.
+288	RESERVED FOR SOFTWARE[6] Reserved for use by system software. Zeroed at system bootstraps or secondary processor starts.
+296	RXTX BUFFER AREA Used for interprocessor console communication. See Section 2.4.
+464	RESERVED Reserved for Digital; SBZ.

[3]Initialized by the console for the primary at all system bootstraps (cold or warm) and for a secondary prior to processor start.

[4]Set by the console at all processor halts.

[6]May by modified by system software for a secondary prior to processor start.

Table 2–4: Per-CPU State Flags

Bit	Description
0	BOOTSTRAP IN PROGRESS (BIP) [3,5,6] For the primary, this bit indicates that this processor is undergoing a system bootstrap. For a secondary, this bit indicates that a CPU start operation is in progress. Set by the console and cleared by system software. See Sections 3.3.1.4, 3.3.3.6, and 3.4.1.

[3]Initialized by the console for the primary at all system bootstraps (cold or warm) and for a secondary prior to processor start.

[5]May be modified by system software for the primary.

[6]May by modified by system software for a secondary prior to processor start.

Table 2–4 (Cont.): Per-CPU State Flags

Bit	Description
1	RESTART CAPABLE (RC)[3,4,5,6] Indicates that system software executing on this processor is capable of being restarted in the event of a detected error halt, powerfail recovery, or other error condition. Cleared by the console and set by system software. See Sections 3.3.1.4, 3.3.3.6, and 3.4.1.
2	PROCESSOR AVAILABLE (PA)[1,2] This bit indicates that this processor is available for use by system software. The PA bit may differ from the PP bit based on self-test or other diagnostics, or as the result of a console command which explicitly sets this processor unavailable.
3	PROCESSOR PRESENT (PP)[1,2] This bit indicates that this processor is physically present in the configuration.
4	OPERATOR HALTED (OH)[3,4] This bit indicates that this processor is in console I/O mode as the result of explicit operator action. See Section 3.4.8.
5	CONTEXT VALID (CV)[3,6] This bit indicates that the HWPCB in this slot is valid. Set after the console or system software initializes the HWPCB in this slot. See Sections 3.3.1.2 and 3.3.3.
6	PALCODE VALID (PV)[1,2] This bit indicates that this processor's PALcode is valid. Set after PALcode has been successfully loaded and initialized. See Sections 3.3.1.2 and 3.3.3.3.
7	PALCODE MEMORY VALID (PMV) [1,2,6] This bit indicates that this processor's PALcode memory and scratch space addresses are valid. Set after the necessary memory is allocated and the addresses are written into the processor's slot. See Sections 3.3.1.2 and 3.3.3.3.
8	PALCODE LOADED (PL) [1,2,6] This bit indicates that this processor's PALcode image has been loaded into the address given in the processor's slot PALcode memory space address field. See Sections 3.3.1.2 and 3.3.3.3.
15:9	RESERVED; MBZ.

[1]Initialized by the console for primary at cold system bootstrap only. Preserved unchanged by the console at all other times.

[2]Initialized by the console for a secondary at cold system bootstrap only. Preserved unchanged by the console at all other times.

[3]Initialized by the console for the primary at all system bootstraps (cold or warm) and for a secondary prior to processor start.

[4]Set by the console at all processor halts.

[5]May be modified by system software for the primary.

[6]May by modified by system software for a secondary prior to processor start.

Table 2–4 (Cont.): Per-CPU State Flags

Bit	Description
23:16	HALT REQUESTED [3,5,6] Indicates the console action requested by system software executing on this processor. Values include: $Code_{16}$ — Reason 0 — Default (no specific action) 1 — SAVE_TERM/RESTORE_TERM exit 2 — Cold Bootstrap requested 3 — Warm Bootstrap requested 4 — Remain halted (no restart) other — Reserved Set to '0' at system bootstraps and secondary processor starts. May be set to non-zero by system software prior to processor halt and subsequent processor entry into console I/O mode. See Sections 3.4.7 and 3.3.5.
63:24	RESERVED; MBZ.

[3]Initialized by the console for the primary at all system bootstraps (cold or warm) and for a secondary prior to processor start.
[5]May be modified by system software for the primary.
[6]May by modified by system software for a secondary prior to processor start.

2.1.4 Configuration Data Block

Systems may have a Configuration Data Block (CONFIG). The format of the block and whether it exists in a system is implementation-specific. If present, the block must be mapped in the bootstrap address space. The CONFIG Offset in the HWRPB (HWRPB[208]) contains the virtual address offset of the block; if no CONFIG block exists, the offset is zero. The first quadword of a CONFIG block must contain the size in bytes of the block. The second quadword must contain a checksum for the block; the checksum is computed as a 64-bit, 2's complement sum ignoring overflows.

2.1.5 Field Replaceable Unit Table

Systems may have a field replaceable unit (FRU) table. The format of the table and whether it exists in a system is implementation-specific. If present, the table must be mapped in the bootstrap address space. The FRU Table Offset in the HWRPB (HWRPB[216]) contains the virtual address offset of the table; if no FRU table exists, the offset is zero.

See the Fault Management Architecture document.

2.2 Environment Variables

The environment variables provide a simply extensible mechanism for managing complex console state. Such state may be variable length, may change with system software, may change as a result of console state changes, and may be established by the console presentation layer. Environment variables may be read, written, or saved.

An environment variable consists of an identifier (ID) and a byte stream value maintained by the console. There are three classes of environment variables:

1. Common to all implementations: ID = 0 to $3F_{16}$.

   These have meaning to both the console and system software. All Alpha consoles must implement all of these environment variables.

2. Specific to a given console implementation: ID = 40 to $7F_{16}$.

   These have meaning to a given console implementation and system software implementation. Support for these environment variables is optional.

3. Specific to system software: ID = 80 to FF_{16}.

   These have meaning to a given system software application or implementation; the console simply passes these environment variables between the console presentation layer and the target application without interpretation. Support for these environment variables is optional.

\

Optional environment variables, if any, supported by a given console must be detailed in the relevant console implementation specification and registered with the Alpha architecture group. See *Appendix E*. \

The value, format, and size of each environment variable is dependent on the environment variable and the console implementation. The size of an environment variable value is specified in bytes. The byte stream value of most environment variables consists of an ASCII string. Some environment variable values consist of multiple fields, some environment variable values consist of lists. Values are parsed as follows:

1. Each field is delimited by one and only one space " " 20_{16}.
2. Each list element is delimited by one and only one comma "," $2C_{16}$.
3. Any numeric quantities are expressed in hexadecimal.
4. All characters are case-blind and may be expressed in uppercase or lowercase.

Examples of environment variables which have list values are BOOT_DEV, BOOTED_OSFLAGS, and DUMP_DEV.

PROGRAMMING TEXT

For example, BOOT_DEV might consist of "0 4 MSCP,0 1 MOP" and BOOT_OSFLAGS might consist of "7,2,1C".

Appendix E summarizes the format and lengths of the environment variables for each implementation.

System software uses the console environment variable routines to access the environment variables. Each environment variable is identified by an identification number (ID). If the console resolves the ID, the associated byte stream value is returned. The console environment variable routines present system software with a consistent interface to environment variables regardless of the presentation layer and internal console representation. The console operator interacts with the console presentation layer to access environment variables. See Section 1.3 for details.

In a multiprocessor system, the console must ensure that the dynamic state created by the environment variables is common to all processors. It must not be possible for a value observed on a secondary to differ from that observed on the primary or another secondary. This is necessary to support bootstrapping, restarting a processor, and switching the primary.

Some environment variables contain critical state which must be maintained across console initializations and system power transitions. Other environment variables contain dynamic state which must be initialized at console initialization and retained across warm bootstraps. Still others contain dynamic state which is initialized at each system bootstrap. See Section 2.5.2.

Environment variable values which must be maintained across console initializations must be retained in some sort of non-volatile storage. Default values for these environment variables must be set prior to system shipment. Thus, there are three possible values: the dynamic value, the default value retained in non-volatile storage, and the initial default value set in non-volatile storage prior to system shipment. The console need not preserve the initial default value. If console implementation preserves the initial default value, that value is accessible only to the console presentation layer; system software accesses only the dynamic and default (last written) values. The dynamic and default values may differ at any time after console initialization as the result of changes by system software or the console operator.

The internal representation and implementation mechanisms of environment variables is at the complete discretion of the console and is unknown to both system software and the console presentation layer. The realization of the required non-volatile storage is also implementation specific.

Table 2–5 lists the environment variables maintained by the console. Each environment ID is also assigned a symbolic name which is used to reference the environment variable elsewhere in this specification.

Table 2–5: Required Environment Variables

Environment Var ID$_{16}$	Symbol	Description
00		Reserved
01	AUTO_ACTION[1,2]	Console action following an error halt or powerup. Defined values and the action invoked are: – "BOOT" (544F 4F42$_{16}$) bootstrap – "HALT" (544C 4148$_{16}$) halt – "RESTART" (54 5241 5453 4552$_{16}$) restart Any other value causes a halt; The default value when the system is shipped is "HALT" (544C 4148$_{16}$). See Section 3.1.1.
02	BOOT_DEV[2]	Device list used by the last (or currently in progress) bootstrap attempt. The console modifies BOOT_DEV at console initialization and when a bootstrap attempt is initiated by a BOOT command. The value of BOOT_DEV is set from the device list specified with the BOOT command or, if no device list is specified, BOOTDEF_DEV. The console uses BOOT_DEV without change on all bootstrap attempts which are not initiated by a BOOT command. See Section 3.3.1.5. The format is independent of the console presentation layer; registered formats are contained in *Appendix E*.
03	BOOTDEF_DEV[1,2]	Device list from which bootstrapping is to be attempted when no path is specified by a BOOT command. See Section 3.3.1.5. The format follows BOOT_DEV. The default value when the system is shipped indicates a valid implementation-specific device or NULL 00$_{16}$.
04	BOOTED_DEV[4]	Device used by the last (or currently in progress) bootstrap attempt. Value is one of the devices in the BOOT_DEV list. See Section 3.3.1.5. The format is independent of the console presentation layer; registered formats are contained in *Appendix E*.
05	BOOT_FILE[1,2]	Filename to be used when a bootstrap requires a filename and when the bootstrap is not the result of a BOOT command or when no filename is specified on a BOOT command. The console passes the value between the console presentation layer and system software without interpretation; the value is preserved across warm bootstraps. The default value when the system is shipped is NULL 00$_{16}$.

[1]Non-volatile. The last value saved by system software or set by console commands is preserved across system initializations, cold bootstraps, and long power outages.

[2]Warm non-volatile. The last value set by system software is preserved across warm bootstraps and restarts.

[4]Read-only. The variable cannot be modified by system system software or console commands.

Table 2–5 (Cont.): Required Environment Variables

Environment Var ID_{16}	Symbol	Description
06	BOOTED_FILE[4]	Filename used by the last (or currently in progress) bootstrap attempt. The value is derived from BOOT_FILE or the the current BOOT command. The console passes the value between the console presentation layer and system software without interpretation.
07	BOOT_OSFLAGS[1,2]	Additional parameters to be passed to system software when the bootstrap is not the result of a BOOT command or when none are specified on a BOOT command. The console preserves the value across warm bootstraps and passes the value between the console presentation layer and system software without interpretation. The default value when the system is shipped is NULL 00_{16}.
08	BOOTED_OSFLAGS[4]	Additional parameters passed to system software during the last (or currently in progress) bootstrap attempt. The value is derived from BOOT_OSFLAGS or the current BOOT command. The console passes the value between the console presentation layer and system software without interpretation.
09	BOOT_RESET[1,2]	Indicates whether a full system reset is performed in response to an error halt or BOOT command. Defined values and the action invoked are: – "OFF" (46 $464F_{16}$) warm bootstrap, no full system reset is performed. – "ON" ($4E4F_{16}$) cold bootstrap, a full system reset is performed. See Sections 3.3.1 and 3.3.2. The default value when the system is shipped is implementation-specific.
0A	DUMP_DEV[1,2]	Device used to write operating system crash dumps. The format follows BOOTED_DEV and is independent of the console presentation layer; registered formats are contained in *Appendix E*. The value is preserved across warm bootstraps. The default value when the system is shipped indicates an implementation-specific device or NULL 00_{16}.

[1]Non-volatile. The last value saved by system software or set by console commands is preserved across system initializations, cold bootstraps, and long power outages.

[2]Warm non-volatile. The last value set by system software is preserved across warm bootstraps and restarts.

[4]Read-only. The variable cannot be modified by system system software or console commands.

Table 2–5 (Cont.): Required Environment Variables

Environment Var ID_{16}	Symbol	Description
0B	ENABLE_AUDIT[1,2]	Indicates whether audit trail messages are to be generated during bootstrap. Defined values and the action invoked are: – "OFF" (46 $464F_{16}$). Audit trail messages suppressed. – "ON" ($4E4F_{16}$). Audit trail messages generated. The default value when the system is shipped is "ON" ($4E4F_{16}$.)
0C	LICENSE[1,4]	Software license in effect. The value is derived in an implementation-specific manner during console initialization. Defined values and (optional) software interpretation are: – "MU" ($554D_{16}$) multiple user system. – "SU" (5553_{16}) single user system. \ Note that the mechanism used to derive the value of LICENSE should NOT be documented in customer-available literature. \
0D	CHAR_SET[1,2]	Current console terminal character-set encoding. Defined values are given in Table 2–7. The default value when the system is shipped is determined by the manufacturing site.
0E	LANGUAGE[1,2]	Current console terminal language. Defined values are given in Table 2–6. The default value when the system is shipped is determined by the manufacturing site.
0F	TTY_DEV [1,2,4]	Current console terminal unit. Indicates which entry of the CTB Table corresponds to the actual console terminal. The value is preserved across warm bootstraps. The default value is "0" 30_{16}.
10–3F		Reserved for Digital.
40–7F		Reserved for console implementation use.
80–FF		Reserved for system software use.

[1]Non-volatile. The last value saved by system software or set by console commands is preserved across system initializations, cold bootstraps, and long power outages.

[2]Warm non-volatile. The last value set by system software is preserved across warm bootstraps and restarts.

[4]Read-only. The variable cannot be modified by system system software or console commands.

Table 2–6: Supported Languages

$LANGUAGE_{16}$	Language	Character-Set	GETC Bytes
0	none (cryptic)	ISO-LATIN–1	1
30	Dansk	ISO-LATIN–1	1
32	Deutsch	ISO-LATIN–1	1
34	Deutsch (Schweiz)	ISO-LATIN–1	1
36	English (American)	ISO-LATIN–1	1
38	English (British/Irish)	ISO-LATIN–1	1
3A	Espanol	ISO-LATIN–1	1
3C	Francais	ISO-LATIN–1	1
3E	Francais (Canadian)	ISO-LATIN–1	1
40	Francais (Suisse Romande)	ISO-LATIN–1	1
42	Italiano	ISO-LATIN–1	1
44	Nederlands	ISO-LATIN–1	1
46	Norsk	ISO-LATIN–1	1
48	Portugues	ISO-LATIN–1	1
4A	Suomi	ISO-LATIN–1	1
4C	Svenska	ISO-LATIN–1	1
4E	Vlaams	ISO-LATIN–1	1
other	reserved	TBD	TBD

Table 2–7: Supported Character Sets

$CHAR\_SET_{16}$	Character-Set
0	ISO-LATIN–1
other	TBD

2.3 Console Callback Routines

System software can access certain system hardware components through a set of callback routines provided by the Alpha console. These routines give system software an architecturally consistent and relatively simple interface to those components.

All of the console callback routines may be used by system software when the operating system has only restricted functionality, such as during bootstrap or crash. When invoked in this context, the console may assume full control of system platform hardware. Some of the console callback routines may be used by system software

when the operating system is fully functional. Such usage imposes constraints on the console implementation.

All routines must be called by system software executing in kernel mode. All routines require that the HWRPB and the per-CPU, CTB, and CRB offset blocks are virtually mapped and kernel read/write accessible. If these conditions are not met, the results are UNDEFINED. Some of the routines execute correctly only at or above certain IPLs.

The routines must never modify any processor registers except those explicitly indicated by the routine descriptions.

2.3.1 System Software Use of Console Callback Routines

Those console callback routines which are intended for use while the operating system is fully functional execute in the unmodified context of that operating system. The console must not usurp operating system control of system platform hardware. These routines must:

1. Not alter the current IPL.
2. Not alter the current execution mode.
3. Not disable or mask interrupts.
4. Not alter any registers except as explicitly defined by the routine interface.
5. Not alter the existing memory management policy.
6. Not usurp any existing interrupt mechanisms.
7. Be interruptable.
8. Ensure timely completion.

Once the operating system is bootstrapped, the console must not reclaim resources transferred to that operating system. This includes both the issuing and servicing of I/O device interrupts, interprocessor interrupts, and exceptions.

It is the responsibility of the console implementation to ensure that these console callback routines may be invoked at multiple IPLs, may be interrupted, and may be invoked by multiple system software threads. The operation of these routines must appear to be atomic to the calling system software even if that software thread is interrupted. See Section Section 2.5.3.1.

In a multiprocessor system, some console routines may be invoked only on the primary processor. A secondary processor may invoke only a subset of these routines and then only under a limited set of conditions. These conditions are explicitly stated in the routine descriptions; if violated, the results are UNDEFINED.

2.3.2 System Software Invocation of Console Callback Routines

With the exception of the FIXUP routine, all of the routines are accessed uniformly through a common DISPATCH procedure. The target routine is identified by a function code. All console callback routines are invoked using the Alpha standard calling conventions.

Any memory management exceptions generated by incorrect mapping or inaccessibility of console callback routine parameters are serviced by the operating system. This occurs naturally for those console callback routines which are intended for use while the operating system is fully functional; these routines execute in the unmodified context of that operating system. For those routines intended for use only while the operating system has restricted functionality, the DISPATCH routine must ensure that any mapping or accessibility conflicts are resolved prior to permitting the console to gain control of the system platform hardware.

2.3.3 Console Callback Routine Summary

The console callback routines fall into four functional groups:

1. Console terminal interaction.
2. Generic I/O device access.
3. Environment variable manipulation.
4. Miscellaneous.

The hexadecimal function code, name, and function for each routine are summarized in Table 2–8.

Table 2–8: Console Callback Routines

$Code_{16}$	Name	Function Invoked
	Console Terminal Routines	
01	GETC	Get character from console terminal
02	PUTS	Put byte stream to console terminal
03	RESET_TERM	Reset console terminal to default
04	SET_TERM_INT	Set console terminal interrupts
05	SET_TERM_CTL	Set console terminal controls
06	PROCESS_KEYCODE	Process and translate keycode
07-F		reserved

Table 2–8 (Cont.): Console Callback Routines

$Code_{16}$	Name	Function Invoked
	Console Generic I/O Device Routines	
10	OPEN	Open I/O device for access
11	CLOSE	Close I/O device for access
12	IOCTL	Perform I/O device-specific operations
13	READ	Read I/O device
14	WRITE	Write I/O device
15–1F		reserved
	Console Environment Variable Routines	
20	SET_ENV	Set (write) an environment variable
21	RESET_ENV	Reset (default) an environment variable
22	GET_ENV	Get (read) an environment variable
23	SAVE_ENV	Save current environment variables
	Console Miscellaneous Routines	
30	PSWITCH	Switch primary processor
(none)	FIXUP	Remap console callback routines
(none)	DISPATCH	Access console callback routine
other		reserved

All Alpha consoles must implement:

1. All console terminal routines except PROCESS_KEYCODE.
2. All console generic I/O device routines.
3. All environment variable routines except SAVE_ENV.
4. The FIXUP and DISPATCH miscellaneous routines.

The PSWITCH routine is required for all Alpha multiprocessor systems which support dynamic primary switching. See Section 3.4.6.

2.3.4 Console Terminal Routines

Alpha consoles provide system software with a consistent interface to the console terminal, regardless of the physical realization of that terminal. This interface consists of the Console Terminal Block (CTB) Table and a number of console terminal routines. Each CTB contains the characteristics of a terminal device which can be accessed through the console terminal routines; see Section 2.3.8.2.

There is ONLY ONE console terminal. The CTB Table may contain multiple CTBs and the console terminal routines may be used to access multiple terminal devices. Each terminal device is identified by a "unit number" which is the index of its CTB within the CTB Table. The TTY_DEV environment variable indicates the unit, hence the CTB, of the console terminal. The console terminal unit is determined at system bootstrap and cannot be altered by system software. Console terminal device interrupts are delivered at IPL 20 to the primary processor; interrupts can be redirected to a secondary only when switching the primary processor.

The console terminal routines permit system software to access the console terminal in a device-independent way. These routines may be invoked while the operating system is fully functional as well as during operating system bootstrap or crash. All console terminal routines are subject to the constraints given in Section Section 2.3.1. These routines must:

1. Not alter the current IPL or current mode.

   These routines must be invoked in kernel mode at or above the console terminal device IPL 20.

2. Not alter the existing memory management policy.

   All internal pointers must have been remapped by FIXUP.

3. Not block interrupts.

   The operating system must be capable of continuing to receive hardware interrupts at higher IPLs.

4. Be interruptable and re-entrant.

   These routines may be invoked at multiple IPLs and their execution may be interrupted. Note, however, that console terminal callback operations are not necessarily atomic. In the event of re-entrant invocations, it is UNPREDICTABLE whether or not the interrupted operation will fail and characters may be transmitted or received out of order.

The time required for console terminal routines to complete is UNPREDICTABLE; however, a console implementation will attempt to minimize the time whenever possible.

SOFTWARE NOTE

To permit use of these routines by OpenVMS, implementations must limit the execution time to significantly less than the interval clock interrupt period. A return after partial operation completion is preferable to long latency.

When invoking these routines, system software must:

1. Be executing in kernel mode at or above the console terminal device IPL 20.

If these routines are invoked in other modes, their execution causes UNPREDICTABLE operation. If invoked at lower IPLs, their execution causes UNDEFINED operation.

2. Be executing on the primary processor in a multiprocessor configuration.

   If these routines are invoked on secondary processors, their execution causes UNDEFINED operation.

3. Be prepared to service any resulting console terminal interrupts, if enabled.

   System software must provide valid interrupt service routines for the console terminal transmit and receive interrupts. The operating system interrupt service routines must be established prior to enabling interrupts; otherwise the operation of the system is UNDEFINED.

PROGRAMMING NOTE

Any console terminal interrupt service routines established by the console prior to transferring control to operating system software are not transferred to the operating system nor are they remapped by FIXUP. Any console terminal interrupts will be delivered only after the operating system lowers IPL from the console terminal device IPL.

IMPLEMENTATION NOTE

The implementation of console terminal I/O interrupts are specific to system hardware platform. An example of implementation-specific characteristics include console terminal SCB vectors.

2.3.4.1 GETC - Get Character from Console Terminal

Format:

```
char      = DISPATCH ( GETC,unit )
```

Inputs:

GETC	= R16;	GETC function code - 01_{16}
unit	= R17;	terminal device unit number
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

char = R0; returned character and status:

R0<63:61>	'000'	success, character received
	'001'	success, character received, more to be read
	'100'	failure, character not yet ready for reception
	'110'	failure, character received with error
	'111'	failure, character received with error, more to be read
R0<60:48>	device-specific error status	
R0<47:40>	SBZ	
R0<39:32>	terminal device unit number returning character	
R0<31:0>	character read from console terminal	

GETC attempts to read one character from a console terminal device and, if successful, returns that character in R0<31:0>. The character is not echoed on the terminal device. The size of the returned character is from one to four bytes and is a function of the current character-set encoding and language, see Table 2–6. The routine performs any necessary keycode mapping.

For implementations which support multiple directly addressable terminal devices, R17 contains the unit number from which to read the character. If the implementation does not support multiple terminal devices or if the devices are not directly addressable, R17 SBZ. The unit number from which the character was read is returned in R0<39:32>. If the implementation does not support multiple terminal devices, R0<39:32> is returned as zero.

GETC returns character reception status in R0<63:61>. If received characters are buffered by the console terminal, R0<61> is set '1' whenever additional characters are available. If GETC returns a character without error, R0<63:62> is set to '00'. If no character is yet ready, R0<63:62> is set to '10'. If an error is encountered obtaining a character, R0<63:62> is set to '11'; examples of errors during character reception include data overrun or loss of carrier.

When an error is returned by GETC, the contents of R0<31:0> and R0<60:48> depend on the capabilities of the underlying hardware. Implementations in which the hardware returns the character in error must provide that character in R0<31:0>. Additional device-specific error status may be contained in R0<60:48>. See the appropriate CTB description in *Appendix E*.

When appropriate, GETC performs special keyboard operations such as turning on or off keyboard LEDs. Such action is based on the incoming stream of keycodes delivered by the console terminal. See the appropriate device CTB description in *Appendix E* for more details.

The return address indicated by R26 should be mapped and kernel executable.

2.3.4.2 PROCESS_KEYCODE - Process and Translates Keycode

Format:

```
char            = DISPATCH( PROCESS_KEYCODE,unit,keycode,again)
```

Inputs:

PROCESS_KEYCODE	= R16;	PROCESS_KEYCODE function code - 06_{16}
unit	= R17;	terminal device unit number
keycode	= R18;	Keycode to be processed
again	= R19;	'1' if calling again for same keycode '0' otherwise
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

char = R0; translated character and status:

R0<63:61>	'000'	success, character returned
	'101'	failure, more time needed to process keycode
	'110'	failure, device not supported by routine or routine not supported
	'111'	failure, no character - more keycodes needed or illegal sequence encountered
R0<60>	'0'	success in correcting severe error
	'1'	failure in correcting severe error
R0<59:32>	SBZ	
R0<31:0>	translated character	

PROCESS_KEYCODE attempts to translate the keycode contained in R18 and, if successful, returns the character in R0<31:0>. The translation is based on the current character-set encoding, language, and console terminal device state contained in the appropriate CTB. The translated character may be from one to four bytes. For implementations which support multiple terminal devices, R17 contains the unit number of the keyboard; R17 SBZ otherwise.

IMPLEMENTATION NOTE

For ISO-LATIN–1 character-set encoding, PROCESS_ KEYCODE returns a one byte character; see Section 2.5.3.2.1.

PROCESS_KEYCODE returns keycode translation status in R0<63:61>. The processing falls into one of several cases:

1. The keycode, along with previous keycodes if any, translates into a character from the currently selected character-set. In this case, R0<63:61> set to '000'.
2. The keycode, along with previously entered keycodes if any, does not translate into a character from the currently selected character-set. This is because either:
   - there are not yet enough keycodes entered to produce a character in the currently selected character-set
   - the keycodes entered to this point indicate a severe keyboard error status
   - the keycodes entered to this point form an illegal or unsupported keycode sequence In this case, R0<63:61> set to '111'.
3. The console terminal device for which keycode translation is being performed is not supported by the PROCESS_KEYCODE implementation or the console implementation does not support PROCESS_KEYCODE. In this case, R0<63:61> set to '110'.
4. The keycode cannot be processed in a reasonable amount of time; multiple invocations of PROCESS_KEYCODE are necessary. In this case, the routine returns with R0<63:61> set to '101'. The subsequent call(s) should be made with the same keycode in R18 and R19 set to '1'.

IMPLEMENTATION NOTE

It may not be possible for an implementation to perform all the actions associated with special keycodes (such as turning on LEDs) in a timely manner. The PROCESS_KEYCODE routine must return after partial operation completion if necessary. It is the responsibility of the console to ensure that subsequent calls make forward progress. The delay between successive operating system calls is UNPREDICTABLE, although the operating system should attempt to complete the operation in a timely fashion. See Sections 2.3.4 and 2.5.3.1.

In all but the first case, the contents of R0<31:0> are UNPREDICTABLE.

When certain severe keyboard errors are encountered, PROCESS_KEYCODE attempts to correct them by performing special keyboard operations. Those severe errors which may be corrected are device-specific and contained in the terminal device CTB. If an error is encountered and the attempt to correct the error is unsuccessful, R0<60> set to '1'; otherwise R0<60> set to '0'.

The keyboard state recorded in the CTB is updated appropriately as the input stream of keycodes is processed. If appropriate, PROCESS_KEYBOARD may buffer some of the keycodes in the CTB keycode buffer. The supported keyboard state changes are device-specific and are listed in the device CTB.

The return address indicated by R26 should be mapped and kernel executable.

2.3.4.3 PUTS - Put Stream to Console Terminal

Format:

```
wcount     = DISPATCH ( PUTS,unit,address,length )
```

Inputs:

PUTS	= R16;	PUTS function code - 02_{16}
unit	= R17;	terminal device unit number
address	= R18;	virtual address of byte stream to be written
length	= R19;	count of bytes to be written
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

wcount = R0; count of bytes written and status:

R0<63:61>	'000'	success, all bytes written
	'001'	success, some bytes written
	'100'	failure, no bytes written, terminal not ready
	'110'	failure, no bytes written, terminal error encountered
	'111'	failure, some bytes written, terminal error encountered
R0<60:48>	device-specific error status	
R0<47:32>	SBZ	
R0<31:0>	count of bytes written (unsigned)	

PUTS attempts to write a number of bytes to a console terminal device. R18 contains the base virtual address of the memory-resident byte stream; R19 contains its 32-bit size in bytes. The byte stream is written in order with no interpretation or special handling. The count of the bytes transmitted is returned in R0<31:0>.

PROGRAMMING NOTE

For multiple byte character-set encodings, the returned byte count may indicate a partial character transmission.

For implementations which support multiple terminal devices, R17 contains the unit number to which the byte stream is to be written; R17 SBZ otherwise.

PUTS returns byte stream transmission status in R0<63:61>. If only a portion of the byte stream was written, R0<61> is set to '1'. If no error is encountered, R0<63:62> is set to '00'. If no bytes were written because the terminal was not ready, R0<63:62> is set to '10'. If an error is encountered writing a byte, R0<63:62> is set to '11'; examples of errors during byte transmission include data overrun or loss of carrier.

When an error is returned by PUTS, additional device-specific error status may be contained in R0<60:48>. See the appropriate CTB description in *Appendix E* for more details.

Multiple invocations of PUTS may be necessary because the console terminal may accept only a very few bytes in a reasonable period of time.

The output byte stream located by R18 should be mapped and kernel read accessible; the return address indicated by R26 should be mapped and kernel executable.

2.3.4.4 RESET_TERM - Reset Console Terminal to default parameters

Format:

```
status      = DISPATCH ( RESET_TERM,unit )
```

Inputs:

RESET_TERM	= R16;	RESET_TERM function code - 03_{16}
unit	= R17;	terminal device unit number
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

status	= R0;	status:		
		R0<63>	'0'	success, terminal reset
			'1'	failure, terminal not fully reset
		R0<62:0>	SBZ	

RESET_TERM resets a console terminal device and its CTB to their initial, default state. All errors in the CTB are cleared. For implementations which support multiple terminal devices, R17 contains the unit number to be reset; R17 SBZ otherwise.

The CTB describes the capabilities of the terminal device and its initial, default state. Depending on the terminal device type and particular console implementation, other terminal devices may be affected by the routine.

PROGRAMMING NOTE

For example, if multiple terminal units share a common interrupt, that interrupt may be disabled or enabled for all.

If the console terminal is successfully reset, RESET_TERM returns with R0<63> set to '0'. If errors are encountered, the routine attempts to return the console terminal to a usable state and then returns with R0<63> set to '1'.

The return address indicated by R26 should be mapped and kernel executable.

2.3.4.5 SET_TERM_CTL - Set Console Terminal Controls

Format:

```
status          = DISPATCH ( SET_TERM_CTL,unit,ctb )
```

Inputs:

SET_TERM_CTL	= R16;	SET_TERM_CTL function code - 05_{16}
unit	= R17;	terminal device unit number
ctb	= R18;	virtual address of CTB
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

status	= R0;	status:		
		R0<63>	'0'	success, requested change completed
			'1'	failure, change not completed
		R0<62:32>	SBZ	
		R0<31:0>	offset to offending CTB field (unsigned)	

SET_TERM_CTL, if successful, changes the characteristics of a console terminal device and updates its CTB. The changes are specified by fields contained in a CTB located by R18. The characteristics which can be changed, hence the active CTB fields, depend on the console terminal device type; see the appropriate CTB description in *Appendix E*. For implementations which support multiple terminal devices, R17 contains the unit number to be reset; R17 SBZ otherwise.

If the console terminal characteristics are successfully changed, SET_TERM_CTL returns with R0<63> set to '0'. If errors are encountered or if the terminal device does not support the requested settings, the routine attempts to return the device to the previous usable state and then returns with R0<63> set to '1' and R0<31:0> set to the offset of an offending or unsupported field in the CTB located by R18. Regardless of success or failure, the device CTB Table entry always contains the current device characteristics upon routine return. SET_TERM_CTL returns the CTB located by R18 without modification.

The CTB located by R18 should be mapped and kernel read accessible; the return address indicated by R26 should be mapped and kernel executable.

2.3.4.6 SET_TERM_INT - Set Console Terminal Interrupts

Format:

```
status          = DISPATCH ( SET_TERM_INT,unit,mask )
```

Inputs:

SET_TERM_INT	= R16;	SET_TERM_INT function code - 04_{16}		
unit	= R17;	terminal device unit number		
mask	= R18;	bit encoded mask:		
		R18<1:0>	'01'	no change to transmit interrupts
			'00'	disable transmit interrupts
			'1X'	enable transmit interrupts
		R18<7:2>	SBZ	
		R18<9:8>	'01'	no change to receive interrupts
			'00'	disable receive interrupts
			'1X'	enable receive interrupts
		R18<63:10>	SBZ	
arginfo	= R25;	argument information		
retadr	= R26;	return address		
procval	= R27;	procedure value		

Outputs:

status	= R0;	status:		
		R0<63>	'0'	success
			'1'	failure, operation not supported
		R0<62:2>	SBZ	
		R0<0>	'1'	transmit interrupts enabled
			'0'	transmit interrupts disabled
		R0<1>	'1'	receive interrupts enabled
			'0'	receive interrupts disabled

SET_TERM_INT reads, enables, and disables transmit and receive interrupts from a console terminal device and updates its CTB. For implementations which support multiple terminal devices, R17 contains the unit number to be reset; R17 SBZ otherwise.

If the interrupt settings are successfully changed, the routine returns with R0<63> set to '0'. If the terminal device does not support the requested setting, then the routine returns with R0<63> set to '1'.

PROGRAMMING NOTE

For example, a device which has a unified transmit /receive interrupt would would not support a request to enable transmit interrupts while leaving receive interrupts disabled.

Regardless of success or failure, the routine always returns with the previous settings in R0<1:0>. The current state of the interrupt settings can be read without change by invoking SET_TERM_INT with R18<1:0> and R18<9:8> set to '01'.

The return address indicated by R26 should be mapped and kernel executable.

2.3.5 Console Generic I/O Device Routines

The Alpha console provides primitive generic I/O device routines for system software use during the bootstrap or crash process. These routines serve in place of the more sophisticated system software I/O drivers until such time as these drivers can be established. These routines may also be used to access console-private devices which are not directly accessible by the processor.

During the bootstrap process, these routines can be used to acquire a secondary bootstrap program from a system bootstrap device. .!or write messages to a terminal other than the logical console terminal. When the operating system is about to crash, these routines can be used to write dump files.

These routines are NOT intended for use while the operating system is fully functional. These routines may:

1. Alter the current IPL.

   The console may raise, but not lower, the IPL for the duration of the routine execution.

2. Block interrupts.

   These routines may cause any and all interrupts to be blocked or delivered to and serviced by the console for the duration of the routine execution.

3. Block exceptions.

   These routines may cause any and all exceptions to blocked or delivered to and serviced by the console for the duration of the routine execution.

4. Alter the existing memory management policy.

   The console may substitute a console-private (or bootstrap address) mapping for the duration of the routine execution.

   PROGRAMMING NOTE

   The console must resolve any virtually addressed arguments prior to altering the existing memory management policy.

5. Take any length of time for completion.

   The operating system has no timeliness guarantee when invoking these routines. Any operating system timer may have expired by their return. The time necessary for completion is UNPREDICTABLE; however, a console implementation will attempt to minimize the time whenever possible.

Prior to returning to the invoking system software, these routines must restore any altered processor state. These routines must return to the calling system software at the IPL and in the memory management policy of that software.

System software invokes these routines synchronously. When invoking these routines, system software must:

1. Be executing in kernel mode.

   If these routines are invoked in other modes, their execution causes UNPREDICTABLE operation.

2. Be executing on the primary processor in a multiprocessor configuration.

   If these routines are invoked on other processors, their execution causes UNDEFINED operation.

2.3.5.1 CLOSE - Close Generic I/O Device for Access

Format:

```
status      = DISPATCH ( CLOSE,channel )
```

Inputs:

CLOSE	= R16;	CLOSE function code - 11_{16}
channel	= R17;	channel to close
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

status	= R0;	status:		
		R0<63>	'0'	success
			'1'	failure
		R0<62:60>	SBZ	
		R0<59:32>	device-specific error status	
		R0<31:0>	SBZ	

CLOSE deassigns the channel number from a previously opened block storage storage I/O device. The channel number is free to be reassigned. The I/O device must be reopened prior to any subsequent accesses.

CLOSE returns status in R0<63>. If the channel was open and the close is successful, R0<63> is set to '0'; otherwise R0<63> is set to '1' and additional device-specific status is recorded in R0<62:32>.

For magnetic tape devices, CLOSE does not affect the current tape position nor is any rewind of the tape performed.

The return address indicated by R26 should be mapped and kernel executable.

2.3.5.2 IOCTL - Perform Device-specific Operations

Format:

```
count     = DISPATCH ( IOCTL,channel,R18,R19,R20,R21 )
```

Inputs:

IOCTL	= R16;	IOCTL function code - 12_{16}
channel	= R17;	channel number of device to be accessed
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

For Magnetic Tape Devices Only:

operate	= R18;	tape positioning operation: '01' for SKIP to next/previous Inter-Record Gap '02' for SKIP over Tape Mark '03' for REWIND '04' for write Tape Mark
count	= R19;	number of SKIPs to perform (signed)
	= R20-R21	Reserved for future use as inputs

Outputs:

For Magnetic Tape Devices Only:

count	= R0;	number of skips performed and status:	
		R0<63:62>	'00' success
			'10' failure, position not found
			'11' hardware failure
		R0<61:60>	SBZ
		R0<59:32>	device-specific error status
		R0<31:0>	number of SKIPs actually performed (signed)

IOCTL performs special device-specific operations on I/O devices. The operation performed and the interpretation of any additional arguments passed in R18 - R21 are functions of the device type as designated by the channel number passed in R17.

For magnetic tape devices, the following operations are defined:

1. '01' - IOCTL relocates the current tape position by skipping over a number of inter-record gaps. The direction of the skip and the number of gaps skipped is given by the signed 32-bit count in R19. Skipping with a count of '0' does not change the current tape position. The number of gaps actually skipped is returned in R0<31:0>.
2. '02' - IOCTL relocates the current tape position by skipping over a number of tape marks. The direction of the skip and the number of marks skipped is given by the signed 32-bit count in R19. Skipping with a count of '0' does not change the current tape position. The number of tape marks actually skipped is returned in R0<31:0>.
3. '03' - IOCTL rewinds the tape to the position just after the Beginning-Of-Tape (BOT) marker. R0<31:0> is returned as SBZ.
4. '04'- IOCTL writes a tape mark starting at the current position. R0<31:0> is returned as SBZ.

IOCTL returns magnetic tape operation status in R0<63:62>. If the operation was successful, R0<63:62> is set to '00'. If the tape positioning was not successful, the tape is left at the position where the error occurred and R0<63:62> is set to '10'. Tape positioning may fail due to encountering a BOT marker (R18 '01' or '02'), encountering a tape mark (R18 '01'), or running off the end of the tape. If a hardware device error is encountered, the final position of the tape is UNPREDICTABLE and R0<63:62> is set to '11'. In the event of an error, additional device-specific status is recorded in R0<61:32>.

The return address indicated by R26 should be mapped and kernel executable.

2.3.5.3 OPEN - Open Generic I/O Device for Access

Format:

```
channel    = DISPATCH ( OPEN,devstr,length )
```

Inputs:

OPEN	= R16;	OPEN function code - 10_{16}
devstr	= R17;	starting virtual address of byte string which contains the device specification
length	= R18;	length of byte string
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

channel = R0; assigned channel number and status:

R0<63:62>	'00'	success
	'10'	failure, device does not exist
	'11'	failure, error - device cannot be accessed or prepared
R0<61:60>	SBZ	
R0<59:32>	device-specific error status	
R0<31:0>	assigned channel number of device	

OPEN prepares a generic I/O device for use by the READ and WRITE routines. R17 contains the base virtual address of a byte string which specifies the complete device specification of the I/O device. The length of the string is given in R18. The format and contents of the device specification string follows that of the BOOTED_DEV environment variable; see *Appendix E*.

The routine assigns a unique channel number to the device. The channel number is returned in R0 and must be used to reference the device in subsequent calls to the READ, WRITE, and CLOSE routines.

OPEN returns status in R0<63:62>. If the I/O device exists and can be prepared for subsequent accesses, R0<63:62> is set to '00'. If the device does not exist, R0<63:62> is set to '10'. If the device exists, but errors are encountered in preparing the device, R0<63:62> is set to '11' and additional device-specific status is recorded in R0<61:32>. In the latter two failure cases, the channel number returned in R0<31:0> is UNPREDICTABLE.

All console implementations must support at least two concurrently opened generic I/O devices. Additional generic I/O devices may be supported.

PROGRAMMING NOTE

See the relevant console implementation specification and *Appendix E*.

For magnetic tape devices, OPEN does not affect the current tape position nor is any rewind of the tape performed.

Multiple channels cannot be assigned to the same device; the second and any subsequent calls to OPEN fail with R0<63:62> set to '11' and R0<31:0> as UNPREDICTABLE. The status of the first opened channel is unaffected.

The input string located by R17 should be mapped and kernel read accessible; the return address indicated by R26 should be mapped and kernel executable.

2.3.5.4 READ - Read Generic I/O Device

Format:

```
rcount    = DISPATCH ( READ,channel,count,address,block )
```

Inputs:

READ	= R16;	READ function code - 13_{16}
channel	= R17;	channel number of device to be accessed
count	= R18;	number of bytes to be read (should be multiple of the device's record length) (unsigned)
address	= R19;	virtual address of buffer to read data into
block	= R20;	logical block number of data to read (used only by disk devices)
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

rcount = R0; number of bytes read and status:

R0<63>	'0'	success
	'1'	failure
R0<62>	'1'	EOT or Logical End of Device condition encountered
	'0'	otherwise
R0<61>	'1'	illegal record length specified
	'0'	otherwise
R0<60>	'1'	run off end of tape
	'0'	otherwise
R0<59:32>		device-specific error status
R0<31:0>		number of bytes actually read (unsigned)

READ causes data to be read from the generic I/O device designated by the channel number in R17 and written to a memory buffer pointed to by R19. The 32-bit transfer byte count, hence length of the buffer, is contained in R18. The buffer must be quadword aligned, virtually mapped, and resident in physical memory.

READ returns transfer status in R0<63:60> and the number of bytes actually read, if any, in R0<31:0>. If the routine is successful, R0<63> is set to '0'. If an error is encountered accessing the device, R0<63> is set to '1'. Additional device-specific status may be returned in R0<59:32>.

The transfer byte count should be a multiple of the record length of the device. If the specified byte count is not a multiple of the record length, R0<61> is set to '1'. If the count exceeds the record length, the count is rounded down to the nearest multiple of the record length and READ attempts to read that number of bytes. If the record length exceeds the count, it is UNPREDICTABLE whether READ attempts to access the device. If no read attempt is made, R0<63> is set to '1'.

For magnetic tape devices, READ does not interpret the tape format nor differentiate between ANSI formatted and unformatted tapes. The routine simply reads the requested transfer byte count starting at the current tape position. READ terminates when either:

1. The specified number of bytes have been read. In this case, R0<63:60> is set to '0000'.
2. An inter-record gap is encountered. In this case, the tape is positioned to the next position after the gap and R0<63:60> is set to '0000'.
3. A tape mark is encountered. In this case, tape is positioned to the next position after the tape mark and R0<63:60> is set to '0100'. (Note that after calling READ and finding a tape mark, the caller can determine if the logical End-Of-Volume or an empty file section has been found by calling READ again. The condition exists if the second READ returns with zero bytes read and a tape mark found.)
4. The routine runs off the end of tape. In this case, R0<63:60> is set to '1001'.

READ ignores End-Of-Tape (EOT) markers.

For disk devices, READ does not understand the file structure of the device. The routine simply reads the requested transfer byte count starting at the logical block number specified by R20. The transfer continues until either the specified number of bytes has been read or the last logical block on the device has been read. If the logical end of the device is encountered, then R0<63:62> is set to '01'.

For network devices, READ interprets and removes any device-specific or protocol-specific packet headers. If a packet has been received, the remainder of the packet is copied into the specified buffer. If a packet has not been received, the routine returns with R0<31:0> set to '0'. Only those network packets which are specifically addressed to this system and are of the specified protocol type are returned; broadcast packets are not returned. The actual packet size is dependent on the device and protocol; the characteristics of the network device and protocol are specified at the time of the channel OPEN.

The buffer pointed to by R19 should be mapped and kernel write accessible; the return address indicated by R26 should be mapped and kernel executable.

2.3.5.5 WRITE - Write Generic I/O Device

Format:

```
wcount     = DISPATCH ( WRITE,channel,count,address,block )
```

Inputs:

WRITE	= R16;	WRITE function code - 14_{16}
channel	= R17;	channel number of device to be accessed
count	= R18;	number of bytes to be written (should be multiple of the device's record length) (unsigned)
address	= R19;	virtual address of buffer to read data from
block	= R20;	logical block number of data to be written (used only by disk devices)
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

wcount = R0; number of bytes written and status:

R0<63>	'0'	success
	'1'	failure
R0<62>	'1'	EOT or Logical End of Device condition encountered
	'0'	otherwise
R0<61>	'1'	illegal record length specified
	'0'	otherwise
R0<60>	'1'	if run off end of tape
	'0'	otherwise
R0<59:32>	device-specific error status	
R0<31:0>	number of bytes actually written (unsigned)	

WRITE causes data to be written to the generic I/O device designated by the channel number in R17 and read from to a memory buffer pointed to by R19. The 32-bit transfer byte count, hence length of the buffer, is contained in R18. The buffer must be quadword aligned, virtually mapped, and resident in physical memory.

WRITE returns transfer status in R0<63:60> and the number of bytes actually written, if any, in R0<31:0>. If the routine is successful, R0<63> is set to '0'. If an error is encountered accessing the device, R0<63> is set to '1'. Additional device-specific status may be returned in R0<59:32>.

The transfer byte count should be a multiple of the record length of the device. If the specified byte count is not a multiple of the record length, R0<61> is set to '1'. If the count exceeds the record length, the count is rounded down to the nearest multiple of the record length and WRITE attempts to write that number of bytes. If the record length exceeds the count, it is UNPREDICTABLE whether WRITE attempts to access the device. If no write attempt is made, R0<63> is set to '1'.

For magnetic tape devices, WRITE does not interpret the tape format nor differentiate between ANSI formatted and unformatted tapes. The routine simply writes the requested transfer byte count starting at the current tape position. WRITE terminates when either:

1. The specified number of bytes have been written without detecting an End-Of-Tape (EOT) marker. In this case, R0<63:60> is set to '0000'.
2. The specified number of bytes have been written and an End-Of-Tape (EOT) marker was detected. In this case, R0<63:60> is set to '0100'.
3. The routine runs off the end of tape. In this case, R0<63:60> is set to '1001'.

For disk devices, WRITE does not understand the file structure of the device. The routine simply writes the requested transfer byte count starting at the logical block number specified by R20. The transfer continues until either the specified number of bytes has been written or the last logical block on the device has been written. If the logical end of the device is encountered, then R0<63:62> is set to '01'.

For network devices, WRITE appends any device-specific or protocol-specific headers. The routine transmits the specified requested transfer bytes with the proper network protocol over the appropriate network. The actual packet size is dependent on the device and protocol; the characteristics of the network device and protocol are specified at the time of the channel OPEN.

The buffer pointed to by R19 should be mapped and kernel write accessible; and the return address indicated by R26 should be mapped and kernel executable.

2.3.6 Console Environment Variable Routines

System software accesses the environment variables indirectly through console callback routines. These routines may be invoked while the operating system is fully functional as well as during operating system bootstrap or crash. The GET_ENV, SET_ENV, and RESET_ENV routines are subject to the constraints given in Section 2.3.1. These routines must:

1. Not alter the current IPL or current mode.

   These routines must be invoked in kernel mode.

2. Not alter the existing memory management policy.

   All internal pointers must be remapped by FIXUP.

3. Not block interrupts.

   The operating system must be capable of continuing to receive hardware and software interrupts.

The constraints on SAVE_ENV differ; see Section 2.3.6.3.

The time necessary for these routines to complete is UNPREDICTABLE; however, a console implementation will attempt to minimize the time whenever possible.

SOFTWARE NOTE

To permit use of these routines by OpenVMS, implementations must limit the execution time to significantly less than the interval clock interrupt period.

The console implementation must ensure that any access to an environment variable is atomic. The console implementation must resolve multiple competing accesses by system software as well as competing accesses by system software and the console presentation layer. See Section 2.5.3.1.

When invoking these routines, system software must be executing in kernel mode. If these routines are invoked in other modes, their execution causes UNPREDICTABLE operation.

These routines may be invoked on both the primary and secondary processors in a multiprocessor configuration. System software is recommended to serialize competing accesses to a given environment variable; a stale value may be returned if GET_ENV is invoked simultaneously with SET_ENV or RESET_ENV.

2.3.6.1 GET_ENV - Get an environment variable

Format:

```
status     = DISPATCH ( GET_ENV,ID,value,length )
```

Inputs:

GET_ENV	= R16;	GET_ENV function code - 22_{16}
ID	= R17;	ID of environment variable
value	= R18;	starting virtual address of byte stream to contain returned value
length	= R19;	number of bytes in byte stream (unsigned)
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

status	= R0;	status:		
		R0<63:61>	'000'	success
			'001'	success, byte stream truncated
			'110'	failure, variable not recognized
		R0<60:32>	SBZ	
		R0<31:0>	count of bytes returned (unsigned)	

GET_ENV causes the value of the environment variable specified by the ID in R17 to be returned in the byte stream specified by the virtual address in R18. The size in bytes of the byte stream is contained in R19.

GET_ENV returns status in R0<63:61>. If the environment variable is recognized, R0<63:62> is set to '00', its current value is copied into the byte stream, and R0<31:0> is set to the number of bytes copied. If the value must be truncated, R0<61> is set to '1'. If the variable is not recognized, R0<63:61> is set to '110' and R0<31:0> is set to '0'.

The byte stream indicated by R18 should be mapped and kernel write accessible; the return address indicated by R26 should be mapped and kernel executable.

2.3.6.2 RESET_ENV - Reset an environment variable

Format:

```
status      = DISPATCH ( RESET_ENV,ID,value,length )
```

Inputs:

RESET_ENV	= R16;	RESET_ENV function code - 21_{16}
ID	= R17;	ID of environment variable
value	= R18;	starting virtual address of byte stream to contain returned value
length	= R19;	number of bytes in byte stream (unsigned)
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

status = R0; status:

R0<63:61>	'000'	success
	'001'	success, byte stream truncated
	'100'	failure, variable read-only
	'101'	failure, variable read-only, byte stream truncated
	'110'	failure, variable not recognized
R0<60:32>	SBZ	
R0<31:0>	count of bytes returned (unsigned)	

RESET_ENV causes the environment variable specified by the ID in R17 to be reset to the system default value and that default value to be returned in the byte stream specified by the virtual address in R18. The size in bytes of the byte stream is contained in R19.

RESET_ENV returns status in R0<63:61>. If the environment variable is successfully reset to the default value, R0<63:62> is set to '00'. If the variable is recognized but read-only, the value is unchanged and R0<63:62> is set to '10'. In both cases, the default value is copied into the byte stream and R0<31:0> is set to the number of bytes copied; if the value must be truncated, R0<61> is set to '1'. If the variable is not recognized, R0<63:61> is set to '110' and R0<31:0> is set to '0'.

The byte stream indicated by R18 should be mapped and kernel write accessible; the return address indicated by R26 should be mapped and kernel executable.

2.3.6.3 SAVE_ENV - Save current environment variables

Format:

```
status      = DISPATCH ( SAVE_ENV )
```

Inputs:

SAVE_ENV	= R16;	SAVE_ENV function code - 23_{16}
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

status	= R0;	status:		
		R0<63:61>	'000'	success, all values saved
			'001'	success, some bytes saved, additional values to be saved
			'110'	failure, routine unsupported
			'111'	failure, error encountered saving values
		R0<60:0>	SBZ	

SAVE_ENV attempts to update the non-volatile storage of those environment variables which must be retained across console initializations and system power transitions. These environment variables are identified as "NV" in Table 2–5.

PROGRAMMING NOTE

For example, SAVE_ENV may cause an EEPROM to be updated. That update may write all "NV" environment variable values to the EEPROM, or may only write those variables which have been modified since the last update or console initialization.

This routine is not subject to the constraints given in Section 2.3.6. The console may usurp operating system control of the system platform hardware, but must restore any such control or altered state prior to return. The console must not service any interrupts or exceptions which are otherwise intended for the operating system.

The non-volatile storage update may take significant time and multiple invocations of SAVE_ENV may be necessary. The time necessary for this routine to complete is UNPREDICTABLE. A console implementation will attempt to minimize the time whenever possible and must return in a timely fashion. The routine must return after partial operation completion if necessary. It is the responsibility of the console

to ensure that subsequent calls make forward progress. The operating system may delay for extended periods between subsequent calls; the console must not rely on timely invocations of SAVE_ENV.

IMPLEMENTATION NOTE

To permit use of these routines by OpenVMS, implementations must limit the execution time to significantly less than the interval clock interrupt period. A return after partial operation completion is preferable to long latency.

SAVE_ENV returns status on the update in R0<63:61>. When the update has successfully completed and all relevant variables have been saved, the routine returns with R0<63:61> set to '000'. If SAVE_ENV returns after only a partial update to ensure timely response, R0<63:61> set to '001'. If an unrecoverable error is encountered, the the routine returns with R0<63:61> set to '111'. The contents of the non-volatile storage are UNDEFINED.

Implementation of SAVE_ENV is optional. If the console does not support SAVE_ENV, the routine returns with R0<63:61> set to '110'.

On a multiprocessor system with an embedded console, the routine must be invoked on each processor in the configuration. See Section 3.7.3.

System software is recommended to ensure that calls to SET_ENV or RESET_ENV are not issued while an update operation is in progress on any processor. It is UNPREDICTABLE whether the updated environment value is saved.

The return address indicated by R26 should be mapped and kernel executable. This routine does not affect the current value of any environment variable maintained by the console.

2.3.6.4 SET_ENV - Set an environment variable

Format:

```
status     = DISPATCH ( SET_ENV,ID,value,length )
```

Inputs:

SET_ENV	= R16;	SET_ENV function code - 20_{16}
ID	= R17;	ID of environment variable
value	= R18;	starting virtual address of byte stream containing value
length	= R19;	number of bytes in byte stream (unsigned)
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

status	= R0;	status:		
		R0<63:61>	'000'	success
			'100'	failure, variable read-only
			'110'	failure, variable not recognized
			'111'	failure, byte stream exceeds value length
		R0<60:31>	SBZ	
		R0<31:0>	maximum value length (unsigned)	

SET_ENV causes the environment variable specified by the ID in R17 to have the value specified by the byte stream value pointed to by the virtual address in by R18. The size in bytes of the byte stream is contained in R19.

SET_ENV returns status in R0<63:61>. If the environment variable is successfully set to the new value, R0<63:61> is set to '000'. If the variable is not recognized, R0<63:61> is set to '110'. If the variable is read-only, the value is unchanged and R0<63:61> is set to '100'. If the input byte stream exceeds the maximum value length, the value is unchanged and R0<63:61> is set to '111'. In all cases, the maximum value length is returned in R0<31:0>.

The byte stream indicated by R18 should be mapped and kernel read accessible; the return address indicated by R26 should be mapped and kernel executable.

2.3.7 Miscellaneous Routines

2.3.7.1 FIXUP - Fixup virtual addresses in console routines

Format:

```
status      = FIXUP ( NEW_BASE_VA, HWRPB_VA )
```

Inputs:

NEW_BASE_VA	= R16;	New starting virtual address of the console callback routines
HWRPB_VA	= R17;	New starting virtual address of the HWRPB
arginfo	= R25;	argument information
retadr	= R26;	return address
procval	= R27;	procedure value

Outputs:

status	= R0;	status:		
		R0<63>	'0'	success
			'1'	failure
		R0<62:0>	SBZ	

FIXUP adjusts virtual address references in all other console callback routines using the new starting virtual address in R16, the new starting virtual address of the HWRPB in R17, and the current contents of the CRB. See Section 2.3.8.1.2 for a full description of FIXUP usage and functionality.

If FIXUP is successful, it returns with R0<63> set to '0'. If FIXUP is not successful, console internal state has been compromised. The console attempts a cold bootstrap if the state transition in Figure 3–1 indicates a bootstrap and the BOOT_RESET environment variable is set to "ON" ($4E4F_{16}$). Otherwise, the system remains in console I/O mode.

This routine must be called in kernel mode and in the context of the existing memory mapping; otherwise its execution causes UNPREDICTABLE or UNDEFINED operation.

SOFTWARE NOTE

FIXUP is generally called while the bootstrap address space mapping is in effect.

The return address indicated by R26 should be mapped and kernel executable.

2.3.7.2 PSWITCH - Switch Primary Processors

Format:

```
status     = DISPATCH ( PSWITCH,action )
```

Inputs:

PSWITCH	= R16;	PSWITCH function code - 30_{16}		
action	= R17;	action requests:		
		R17<1:0>	'01'	transition from primary
			'10'	transition to primary
			'11'	switch primary
		R17<63:2>	SBZ	
cpu_id	= R18;	new primary CPU ID		
arginfo	= R25;	argument information		
retadr	= R26;	return address		
procval	= R27;	procedure value		

Outputs:

status	= R0;	status:		
		R0<63>	'0'	success
			'1'	failure, operation not supported
		R0<62:0>	implementation-specific error status	

PSWITCH attempts to perform any implementation-specific functions necessary to support primaryness switching. R17 indicates the requested primary transition action. R18 contains the CPU ID (WHAMI IPR) of the new primary.

PSWITCH is invoked by the old primary, the secondary which is to become the new primary, or both. See Section 3.4.6 for a full description of PSWITCH usage, functionality, and error returns.

If PSWITCH is successful, it returns with R0<63> set to '0'. If PSWITCH is unsuccessful for any reason, it returns with R0<63> set to '1' and implementation-specific status in R0<62:0>.

PSWITCH is invoked at IPL 31. The return address indicated by R26 should be mapped and kernel executable.

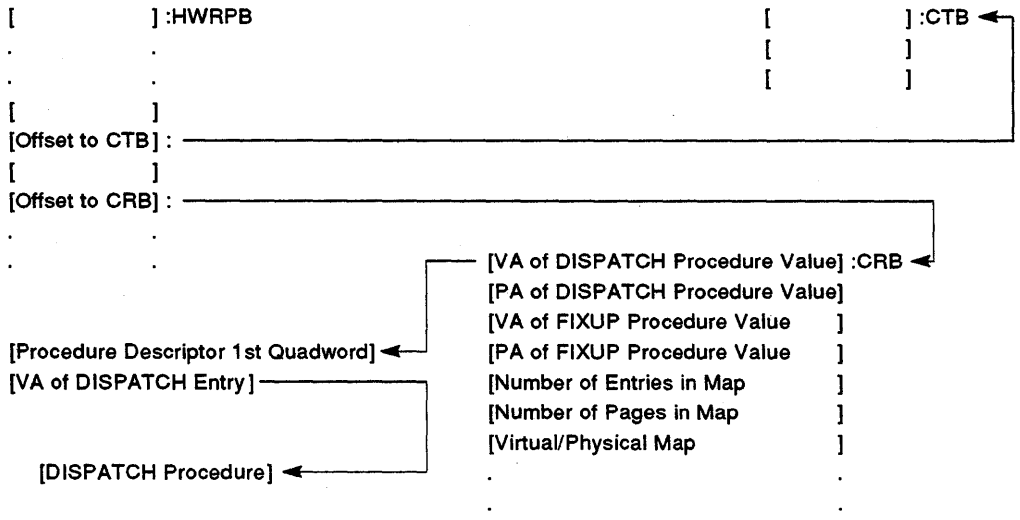
2.3.8 Console Callback Routine Data Structures

The console and system software share two data structures which are necessary for the console callback routines. These are the Console Routine Block (CRB) and the Console Terminal Block (CTB) Table. Both are located by offset fields in the HWRPB as show in Figure 2–4.

The CRB locates all addresses necessary for console callback routine function. The base physical address of the CRB is obtained by adding the CRB OFFSET field at HWRPB[192] to the base physical address of the HWRPB. The CRB format is shown in Figure 2–5 and described in Table 2–9.

The CTB Table contains information necessary to describe the console terminal devices. The base physical address of the CTB Table is obtained by adding the CTB TABLE OFFSET field at HWRPB[184] to the base physical address of the HWRPB. The CTB format is shown in Figure 2–6 and described in Table 2–10.

Figure 2–4: Console Data Structure Linkage



2.3.8.1 Console Routine Block

Prior to transferring control to system software, the console ensures that the console callback routines, console-private data structures, and associated local I/O space locations are mapped into region 0 of initial bootstrap address space. All necessary pages are located by the Console Routine Block (CRB).

Figure 2–5: Console Routine Block

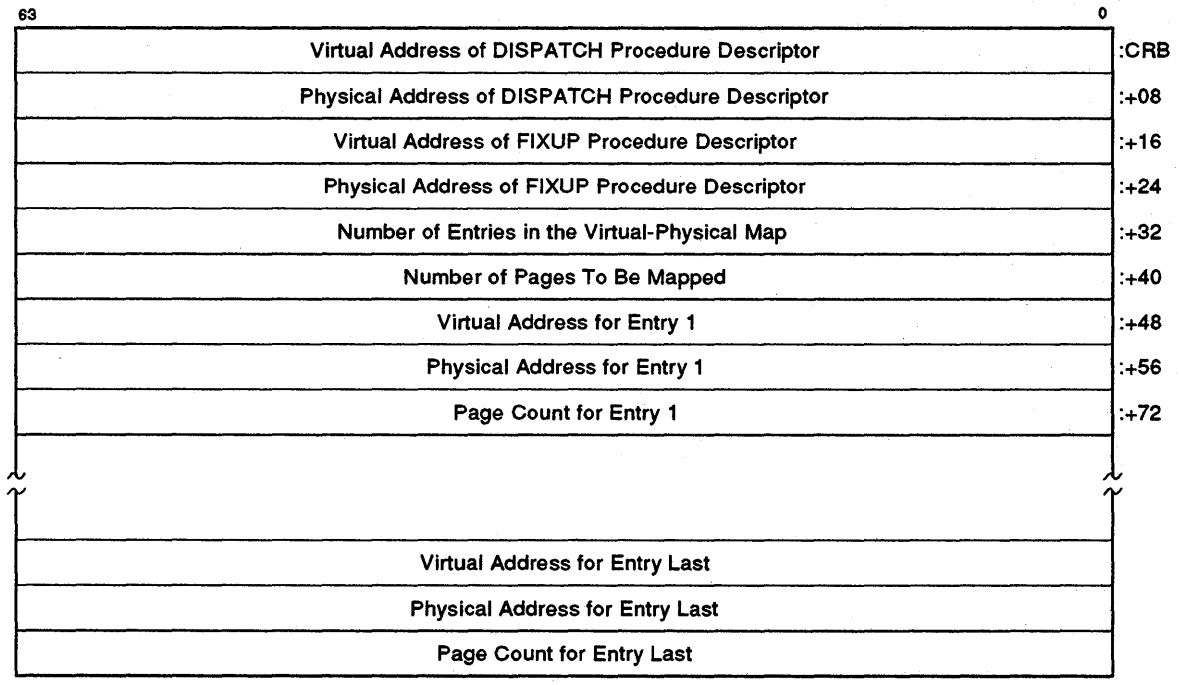


Table 2–9: CRB Fields

Offset	Description
CRB	DISPATCH VA - The virtual address of the procedure descriptor for the DISPATCH procedure.
+08	DISPATCH PA - The physical address of the procedure descriptor for the DISPATCH procedure.
+16	FIXUP VA - The virtual address of the procedure descriptor for the FIXUP procedure.
+24	FIXUP PA - The physical address of the procedure descriptor for the FIXUP procedure.
+32	ENTRIES - The number of entries in the virtual-physical map. Unsigned integer.
+40	PAGES - The total number of physical pages to be mapped. Unsigned integer.
+48	ENTRY - Each entry identifies a collection of physically contiguous pages to be mapped. Each map entry consists three quadwords:

Offset	Name	Description
+00	ENTRY_VA	Base virtual address for entry
+08	ENTRY_PA	Base physical address for entry
+16	ENTRY_PAGES	Number of contiguous physical pages to be mapped. Unsigned integer.

The CRB must be quadword aligned. The DISPATCH and FIXUP addresses must be quadword aligned; all unused bits SBZ. The ENTRY addresses must be page aligned and all unused bits SBZ.

The DISPATCH and FIXUP procedure descriptors located by DISPATCH_PA, DISPATCH_VA, FIXUP_PA and FIXUP_VA must be contained within the pages located by the first virtual-physical map entry.

2.3.8.1.1 Console Routine Block Initialization

Prior to transferring control to system software, the console initializes all fields of the CRB. The console fills in all physical and virtual address fields, the number of entries in the virtual-physical map (ENTRIES), the total number of pages to be mapped (PAGES), and the virtual addresses contained in the procedure descriptors for the DISPATCH and FIXUP procedures[1]. PAGES is the sum of the contents of all ENTRY_PAGES fields.

All addresses are initially mapped within region 0 of the initial bootstrap address space. These addresses include the contents of the CRB and all addresses contained

[1] Recall from the Alpha calling standard, that the second quadword of a procedure descriptor contains the entry address (virtual) of the procedure itself.

within the DISPATCH and FIXUP procedure descriptors. The mapping must permit kernel access with appropriate read/write/execute access. Note that the KRE, KWE, and FOx PTE fields are never subsequently altered by system software. The initial mapping need not be virtually contiguous.

2.3.8.1.2 Console Routine Remapping

When the console transfers control to the system software, the console callback routines may be invoked by the system software without additional setup. All necessary virtual mappings into initial bootstrap address space must be performed by the console prior to transferring control.

The system software may virtually remap the console callback routines. This remapping permits the system software to relocate the routines to virtual addresses other than those assigned in initial bootstrap address space. This relocation requires that the console adjust (or fixup) various internal virtual address references.

The system software invokes the FIXUP routine to enable the console to perform the necessary internal relocations. The FIXUP routine virtually relocates all console routines and adjusts any console-private virtual address pointers such as those used to locate a local I/O device or HWRPB data structure. Note that if system software virtually remaps the HWRPB, FIXUP must be invoked prior to calling any other console callback routine; it is recommended that system software remap both the HWRPB and the console routines together[1]. Calling the console callback routines after the HWRPB has been remapped from its original bootstrap address location results in UNDEFINED operation of the system.

To remap the console callback routines, the system software and the console cooperate as follows:

1. System software must be executing on the primary processor in a multiprocessor system.
2. System software determines the new base virtual address of the HWRPB; this remapping is optional. System software does not perform any remapping of the HWRPB at this step.

   Note that system software need not remap the memory data descriptor table located by HWRPB[200]. See Section 2.1 for a description of the HWRPB and its size.
3. System software determines the new base virtual address of the console callback routines. The CRB entries will be mapped into a set of virtually contiguous pages. The CRB PAGES field (CRB[40]) is used to determine the number of pages that must be mapped. System software does not perform any remapping of the console callback routines at this step.
4. System software passes control to the console by calling FIXUP (NEW_BASE_VA, NEW_HWRPB_VA). NEW_BASE_VA is the new base virtual address as estab-

[1] Note that if the HWRPB is remapped but subsequently returned to its original bootstrap address location, the routines may be successfully invoked after the return of the HWRPB to its original remapping without calling FIXUP.

lished in step 3. HWRPB_VA is the new starting virtual address of the HWRPB as established in step 2.

5. The console first locates the HWRPB, then locates the CRB using the CRB OFFSET field. The console then locates all internal pointers and adjusts them. All linkage sections and other console-internal pointers must be modified. These data structures can be located during FIXUP because the initial bootstrap address space mapping is in effect; any console-internal pointers are valid until modified.

   Note that system software need not remap the optional CONFIG Block or FRU Table located by HWRPB OFFSET fields. If these blocks will be subsequently used by the console, they must be located by console-internal pointers and those pointers must be modified during FIXUP.

   DISPATCH and FIXUP are not uniquely remapped by the system software. The FIXUP must update the DISPATCH and FIXUP procedure descriptors located by CRB[8] and CRB[24]. The physical pages containing the procedure descriptors and the routines themselves must be included in the virtual-physical map.

   Lastly, note that the relative virtual address offsets of the pages located by the entry map are not guaranteed to be retained across the FIXUP. The initial bootstrap address mapping of the physical pages located by the entry map is not required to be virtually contiguous. The system software remapping is required to be virtually contiguous. Any offsets which cross physical pages may have to be modified by FIXUP.

6. The console returns from FIXUP. If the FIXUP was not successful, console internal state has been compromised. The console attempts a cold bootstrap if the state transition in Figure 3–1 indicates a bootstrap and the BOOT_RESET environment variable is set to "ON" ($4E4F_{16}$). Otherwise, the system remains in console I/O mode.

7. System software updates each virtual-physical map entry of the CRB:

   1. The PTE and TB entries corresponding to the range of old virtual address are invalidated using the old ENTRY_VA and ENTRY_PAGES values.

   2. The new starting virtual address is written into the ENTRY_VA. This virtual address is computed by adding the NEW_BASE_VA to the sum of the PAGE_COUNTs of each preceding entry.

   3. New PTEs are constructed for each physical page. The new PTE FOx and protection fields are copied from the original bootstrap address PTE.

PROGRAMMING NOTE

Note that it is the responsibility of the console to judiciously set both the protection and FOx bits in the bootstrap address PTE. In particular, if the console sets the FOE bit, there is no architectural guarantee that the console exception

handler will gain control nor any obvious appropriate response for the operating system handler.

8. System software updates the DISPATCH and FIXUP VAs. The first virtual-physical map entry locates the physical page which contains the DISPATCH and FIXUP procedure descriptors.
9. System software updates all PTEs and invalidates all appropriate TB entries associated with the remapped HWRPB and any remapped OFFSET blocks.

At the completion of this process, the console callback routines are remapped and may again be used by system software. Note that since FIXUP itself is relocated, system software may remap the routines more than once.

2.3.8.2 Console Terminal Block Table

The Console Terminal Block (CTB) Table indicates the current identity and characteristics of each console terminal device. The CTB Table is the only data structure shared by the console and system software which describes the terminal devices accessible by console callback routines.

The CTB Table contains an array of CTBs. Each CTB is a quadword-aligned structure with format as shown in Figure 2–6 and described in Table 2–10. The index of the CTB in the CTB Table is the unit number of the terminal device. The CTB format consists of two parts: a header and a device-specific segment. The format of the header is common to all CTBs; the format of the device-specific segment is dependent on the unique device type. *Appendix E* contains the specification of all registered CTB formats.

There is ONLY ONE console terminal. The console terminal unit is selected by the console presentation layer prior to bootstrapping the operating system; see Section 1.3. Once the operating system is bootstrapped, the console terminal unit should not be changed by the console presentation layer. Any attempt to do so results in UNDEFINED operation of the console. Specifically, if the console presentation layer halts the operating system, alters the console terminal unit, then restarts or continues operating system execution, the operation of the console is UNDEFINED. The console terminal unit is identified by the TTY_DEV environment variable.

During console initialization, the console:

1. Locates all console terminal devices.
2. Selects the console terminal.
3. Builds a CTB for each.
4. Initializes the CTB OFFSET field of the HWRPB.
5. Initializes each console terminal device.
6. Records the default state of each console terminal device in its CTB.
7. Records the unit number of the console terminal in the TTY_DEV environment variable.

Whenever the console changes the state of a console terminal device, the console must update its CTB to reflect the change. The console may record extended status on character transfers (GETC/PUTS) in the CTB.

System software uses the CTB to determine console terminal device characteristics. System software never directly modifies the contents of a CTB; such modifications can result in UNDEFINED operation of the console terminal device either as the result of a subsequent call to a console terminal routine or as the result of a console internal need to access a console terminal device (e.g. as the result of a halt). System software calls the SET_TERM_CTL console terminal routine to change console terminal device characteristics.

Figure 2–6: Console Terminal Block

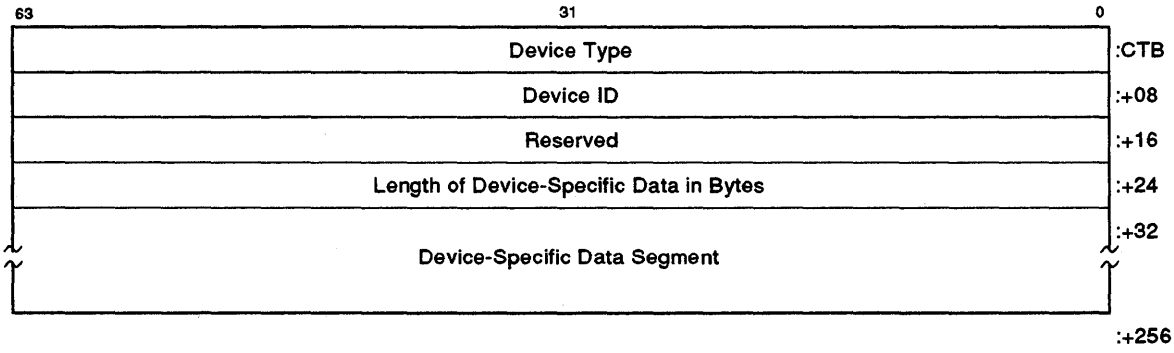


Table 2–10: CTB Fields

Offset	Description
CTB	DEVICE TYPE - Console terminal device type and format of the device-specific segment. Defined device types are:

Type	Description
0	No console present
1	Detached service processor
2	Serial line UART
3	Graphics display with LK keyboard connected to serial line UART
other	Reserved

Offset	Description
+08	DEVICE ID - The physical device and channel which sends and receives the console terminal stream. This field is necessary for configurations which include multiple-channel devices or multiple single-channel devices. The field has two subfields:

Bits	Description
<63:32>	Device index
<31:0>	Channel index

Offset	Description
	For implementations which support only a single directly-connected console terminal device, this field is set to zero. Note that the device ID is not necessarily related to the console terminal device unit number.
+16	RESERVED - This field is reserved for future expansion and may not be used by the console or system software.
+24	DSD LENGTH - This field specifies the number of bytes in the device-specific data field, DSD.
+32	DSD - This field contains device-specific data associated with the unique console terminal type. Device-specific data may include such parameters as baud rate, flow control is enables, and the current state of the CAPS LOCK key. The DSD field should contain only those items which are must be shared between the console and system software.

2.4 Interprocessor Console Communications

Only those communications between a running processor and a console processor are considered here. Communications paths between running processors are external to the console. Communications paths between console processors are internal to the console. See Section 2.5.4.

Commands are transmitted from a running primary to a console secondary; messages (and requests) are transmitted from a console secondary to a running primary. Commands and messages are passed via Receive (RX) and Transmit (TX) buffers

contained in each per-CPU slot of the HWRPB. The use of these buffers is controlled by the Receive Buffer Ready (RXRDY) and Transmit Buffer Ready (TXRDY) flags. Messages consist of the message symbol as given in Table 1–1.

PROGRAMMING NOTE

For example, "?PALREQ?" is passed to request PALcode loading.

Commands use the command syntax given in Section 1.3.

The transmit and receive buffers are named from the point of view of the console secondary. The console secondary receives commands in the RX buffer and transmits messages in the TX buffer.

2.4.1 Interprocessor Console Communications Flags

The Receive Buffer Ready (RXRDY) and Transmit Buffer Ready (TXRDY) flags are used to control the interprocessor console communications. The RXRDY and TXRDY flags are gathered into bitmasks in the HWRPB at HWRPB[296] and HWRPB[304] respectively. The TXRDY bitmask allows a running primary to quickly determine which, if any, of the console secondaries are trying to send messages.

The running primary sets the appropriate RXRDY flag to indicate to the receiving console secondary that a command is contained in the secondary's RX buffer. The secondary is assumed to be polling its RXRDY flag. The RXRDY flag is cleared by the secondary after the command has been read from the RX buffer and prior to executing the command.

A console secondary sets its TXRDY flag to indicate to the running primary that a message is contained in the secondary's TX buffer. The console generates an interprocessor interrupt to the primary to notify it that a message is ready. System software clears the TXRDY flag after the message has been read from the TX buffer and prior to processing the message.

IMPLEMENTATION NOTE

The TXRDY bitmask minimizes interprocessor interrupt service overhead by reducing the number of required memory lookups.

2.4.2 Interprocessor Console Communications Buffer Area

Each per-CPU slot of the HWRPB includes an RXTX Buffer Area which provides the communications path between processors. The buffer area is controlled by the RXRDY and TXRDY flags. The format is shown in Figure 2–7 and described in Table 2–11.

Figure 2–7: Inter-Console Communications Buffer

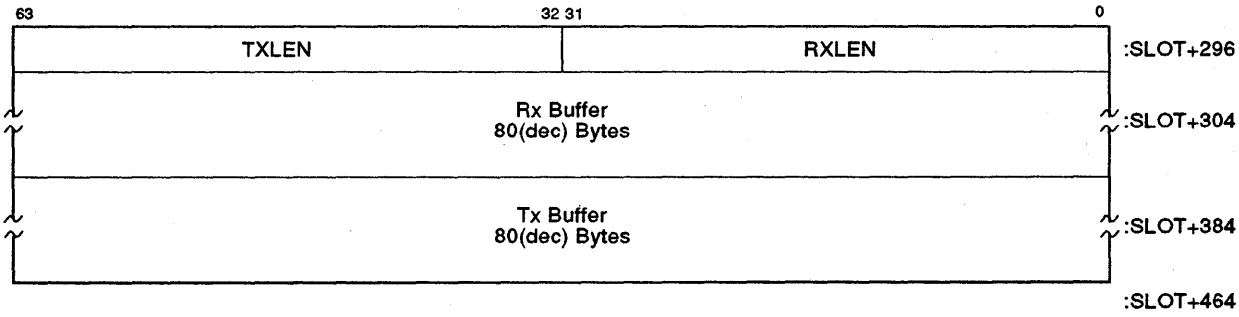


Table 2–11: Inter-Console Communications Buffer Fields

Offset	Description
SLOT+296	RXLEN - If the bit corresponding to this processor is set in the RXRDY bitmask at HWRPB[296], the RXLEN field contains the length in bytes of the command in the RX buffer.
+300	TXLEN - If the bit corresponding to this processor is set in the TXRDY bitmask at HWRPB[304], the TXLEN field contains the length in bytes of the message in the TX buffer.
+304	RX BUFFER - Buffer used by this console secondary to receive a command from the running primary. Only command data is passed through this buffer; a console secondary does not receive messages from the running primary. Commands must end with "<CR><LF>" ($0A0D_{16}$).
+384	TX BUFFER - Buffer used by this console secondary to transmit a message to the running primary. Only message data is passed through this buffer; a console secondary does not send commands to the running primary. Messages must end with with the console secondary's prompt, "<CR><LF>Pnn>>>" (3E3E 3Enn nn50 $0A0D_{16}$).

2.4.3 Sending a Command to a Secondary

The running primary manipulates the secondary's RXRDY flag and RX buffer in the following manner to send a command to a console secondary. In the sequence, the console secondary is assumed to have CPU ID = "N".

PROGRAMMING NOTE

The RXRDY flag is a software lock variable; the primary and the secondary must use LDQ_L/STQ_C instructions to set and clear bit "N". See *Common Architecture, Chapter 5*.

1. The primary examines bit "N" of the RXRDY bitmask. If the bit is clear, proceed to step 3.
2. The primary polls bit "N" of the RXRDY bitmask until clear or until some timeout is reached. If a timeout occurs, system software reports an error and takes appropriate action.
3. The primary moves the text of the desired console command into the RX buffer in the secondary's HWRPB slot (the "Nth" per-CPU slot).
4. The primary sets the length of the command into the RXLEN field in the secondary's HWRPB slot (the "Nth" per-CPU slot).
5. The primary sets bit "N" of the RXRDY bitmask to indicate there is a command waiting.
6. The secondary is assumed to be polling bit "N" of the RXRDY bitmask.
7. When the secondary notices that bit "N" of the RXRDY bitmask is set, it removes the command from its RX buffer.
8. The secondary clears bit "N" of the RXRDY bitmask, indicating that its RX buffer is again available.
9. The secondary attempts to process the command.

2.4.3.1 Sending a Message to the Primary

The console secondary manipulates its TXRDY flag and TX buffer in the following manner to return a message to the running primary. Again, the console secondary is assumed to have CPU ID = "N".

PROGRAMMING NOTE

The TXRDY flag is a software lock variable; the primary and the secondary must use LDQ_L/STQ_C instructions to set and clear bit "N". See *Common Architecture, Chapter 5*.

1. The secondary examines bit "N" of the TXRDY bitmask. If the bit is clear, then proceed to step 3.
2. The secondary polls this bit until it clears or until a long timeout occurs. (See step 7.)
3. The secondary moves the text of its response message into the TX buffer in the secondary's HWRPB slot (the "Nth" per-CPU slot).

4. The secondary sets the length of the message into the TXLEN field in the secondary's HWRPB slot (the "Nth" per-CPU slot).
5. The secondary sets bit "N" of the TXRDY bitmask to indicate there is a message waiting.
6. The secondary issues an interprocessor interrupt to the primary. This is always done; the primary need not poll for bits in the TXRDY bitmask.
7. The secondary polls the TXRDY bitmask until bit "N" clears or until a long timeout expires. This prevents the secondary from performing any action which might cause the message to be lost before the primary can process it.

PROGRAMMING NOTE

The secondary may be restarted once it has transmitted the error halt message to the primary. However, it must wait for the primary to have a reasonable chance to respond to the interprocessor interrupt and process the message before the restart proceeds since that message is important visible evidence of the error halt condition. On the other hand, the secondary shouldn't wait forever for the primary to respond since the primary may be affected by the same condition that caused the secondary to error halt. Hence, the need for a timeout that is of reasonable length.

8. As a result of the interprocessor interrupt, the primary eventually checks for console messages by examining the TXRDY bitmask. The primary notices that bit "N" of the TXRDY bitmask is set.
9. The primary removes the message from the TX buffer.
10. The primary clears bit "N" of the TXRDY bitmask, indicating that the TX buffer is again available.
11. The primary attempts to process the message.

2.5 Implementation Considerations

2.5.1 Serial Number and Revision Fields

The system serial number and revision fields must be distinct from the processor serial number and revision fields. In particular, on multiprocessing systems, the system fields must not be simply replicated from the fields of the primary processor. The system fields must be constant regardless of which processor serves as primary and must have persistence across processor failures and/or replacement.

This is necessary to permit application software to determine the system identity in a dependable fashion. An example of such application software is the system error log. If the system serial number were tied to a given processor, the error log would report a different serial number if that processor is later unavailable for any reason.

2.5.2 Console Environment Variables

While the HWRPB is the primary means of communication between the console and system software, there are cases for which it is ill-suited:

- Because the HWRPB resides in main memory, it cannot preserve certain critical components of the console state across powerfails. This state must be held by a mechanism that can survive powerfails. This state includes the necessary information to reboot system software after a powerfail.
- The structure of the HWRPB is too rigid for the direct inclusion of variable-length console state which may be added after system software bootstraps. Support for variable-length state through the HWRPB would require the HWRPB to contain pointers to the actual state. The usage of memory reserved for this actual state would require negotiation between the console and system software.
- There is a need for the console presentation layer to establish environment parameters which affect the bootstrapping of system software. Many of these parameters are set only once, but stay in effect across subsequent bootstraps and in some cases across the powering down and up of the system. This requirement is in effect today on both VAXes and DECsystems. The number, format, size, and legal values of these parameters are established by system software and may change from one revision to the next; they cannot be predicted by the console. Using the HWRPB to share these parameters between the console presentation layer and system software is awkward at best.

The Alpha console solves the requirements of the above cases with one unified approach: environment variables.

The console environment variable routines must present a consistent interface to environment variables regardless of the presentation layer and regardless of the internal representation. For example, an ISO-LATIN–1 French console presentation layer could accept and display text for the BOOT_RESET environment variable as "marche" or "oui" for $4E4F_{16}$ and "arrjt" or "non" for 46 $464F_{16}$ provided that the values of $4E4F_{16}$ or 46 $464F_{16}$. are returned to system software by GET_ENV.

Console implementations are recommended to maintain a memory-resident copy of all non-volatile environment variables to ensure that the access time to these variables remains within acceptable bounds. Examples of non-volatile storage media for environment variables include EEPROM, Flash ROM, and a console-private I/O device.

A need to distinguish between environment variable values which are static across console initializations from those which are static across system bootstraps was necessary to support OpenVMS Alpha host-based shadow set bootstraps. This separation permits the operating system to "temporarily" change the environment variables which govern bootstrapping. During shadow set state transitions, the operating system system disk must be a known valid shadow set member and that member or members cannot be determined until after the initial bootstrap process has completed and the system initialization has begun. Temporarily altering the environment variables enables the operating system to reorder the console bootstrap device list to ensure that the next, rapidly ensuing, bootstrap attempt will use a known

valid shadow set member. Such an alteration is known to be transient and should not affect the normal bootstrap controls.

Console implementation-specific or system software specific environment variable may be volatile or non-volatile. The nature of these environment variables is at the discretion of the console implementation.

2.5.3 Console Callback Routines

2.5.3.1 System Software Use of Console Callback Routines

For those console callback routines intended for use while the operating system is fully functional, the console implementation must ensure that system software can invoke those routines at multiple IPLs and that the execution may be interrupted. The console implementation must ensure that internal console state is not corrupted by conflicting requests by the console presentation layer and system software.

Consider the case of an operating system debugger which gains control of the processor during the execution of a console terminal routine invoked by system software executing at a lower IPL. The debugger must be able to access the console terminal. The console implementation may not block the higher IPL call. Note, however, that system software is recommended to serialize such accesses. If routine execution is resumed at the lower IPL, the console need not guarantee that the resumed operation completes correctly. For example, if the routine requires access which is not atomic (for example indirect register access), the console implementation need not ensure that the resulting pattern of accesses do not result in an UNPREDICTABLE condition.

Similarly, consider the case of a system software invocation of SET_ENV which is suspended by a processor halt. If the console presentation layer sets that same environment variable and then continues the system, the new environment variable value must be either that specified by the console presentation layer OR that specified by the system software. The value must not be corrupted even if there is no hardware guarantee of atomicity.

2.5.3.2 Console Terminal Routines

The console terminal routines are intended to provide a consistent interface to the "console terminal device", regardless of the physical realization of that "terminal device". The "console terminal device" may be a physical terminal directly connected to an embedded console by a UART or an graphic application executing on a workstation which is networked to the processor console.

The CTB solves the problem present in previous VAX systems where there are differently formatted data structures describing the same device across the various systems. This increases the burden of the operating system to support new systems. By requiring all Alpha implementations to use the same CTB format for the same device, the burden of supporting new systems by system software is lessened.

A simple example of a multiple-channel controller is a DZ; multiple serial lines, each going to a different external device, share one set of CSRs and device characteristics. Another example of a multiple-channel controller is the DEFNA which supports

multiple, possibly disjoint, Ethernets. A simple example of multiple single-channel controllers is multiple SGECs, each of which connects to a unique Ethernet.

To simplify system software interrupt handling, it is recommended that implementations provide separate console terminal transmit and receive interrupts.

2.5.3.2.1 PROCESS_KEYCODE

PROCESS_KEYCODE is intended for use by system software which must acquire keycodes directly from the console terminal device; PROCESS_KEYCODE translates the keycode into characters of the currently selected character-set. GETC is the normal method used to acquire characters from the console terminal; GETC performs any necessary translation.

CTB information relevant to the translation includes the type of display-keyboard combination and the current keyboard state. In the process of translation, the routine may buffer previously entered keycodes in the CTB.

The supported display-keyboard combinations are specific to the console implementation. Only those combinations which are supported by the console implementation are processed and translated by PROCESS_KEYCODE.

Examples of severe keyboard errors which may be corrected include the LK401 keycodes: OUTPUT ERROR, INPUT ERROR, and TEST MODE ACKNOWLEDGE.

Examples of keyboard state changes include shifting to uppercase keys, enabling CAPS LOCK and lighting the CAPS LOCK LED, and activating output flow control and lighting the HOLD SCREEN LED.

This routine in intended to ease software effort to support graphics workstations in which the console presentation layer shares the workstation screen.

2.5.3.3 Console Block Storage Routines

These routines are provided for operating systems whose primary bootstrap is not large enough to carry the necessary I/O drivers to fetch the system image. This is particularly a problem for ULTRIX, where the primary bootstrap must fit into logical blocks 1 to 15 of the ULTRIX system disk. The console possesses most of the capabilities specified in these block storage routines due to boot device requirements. As such, permitting system software to make use of that functionality seemed both beneficial and simple to provide.

2.5.3.4 FIXUP

When considering how to make the console routines compliant to the Alpha Calling Standard and virtually relocatable, two choices quickly presented themselves.

1. Provide the physical and virtual addresses of the procedure descriptor for each routine, ensure that the descriptor existed, and give the pages necessary to map for it. The resulting relocation would be quite piecemeal and, moreover, did not address the relocation of any necessary routine-private pointers (e.g. local I/O device registers.)

2. Provide a calling standard compliant interface, DISPATCH, give only the virtual and physical addresses for its procedure descriptor, and a (gather-scatter) list of physical pages to be mapped and relocated. The resulting relocation is less piecemeal and addresses the relocation of any necessary routine-private pointers.

Note that both choices still require a virtual address FIXUP routine for relocation.

The DISPATCH procedure and the console routines should be consolidated into a few contiguous physical pages. Implementations should attempt to reduce the necessary CRB mapping entries.

2.5.4 Interprocessor Console Communications

Considering the reasonable combinations of the four processor states, the following communications paths must be provided:

1. Running processor to running processor.

   These paths are external to the console and independent of which is primary or secondary. They are supported by the communications mechanisms within the operating system. These paths are used even when the communications is related to the console. For example, an operating system debugger entered on a secondary is responsible for passing characters to and from the primary, and thus to the console terminal.

2. Running primary to/from console secondary.

   The operating system on the primary must be able to send complete console commands to a console secondary, for example to start a secondary. A console secondary must be able to send messages to the operating system on the running primary, for example when the secondary encounters an error halt. Such messages may be sent by a secondary at any time.

   It is not necessary for a secondary to send commands to the primary, or for the primary to send messages to a secondary.

3. Console primary to/from running secondary.

   It is unclear what communication is necessary along this path. It is likely that whenever the primary halts, the secondaries will eventually block waiting for resources locked by the primary. The console primary will support receiving complete messages from a running secondary.

   NOTE

   All consoles include a mechanism to force a running primary into console I/O mode. Specific secondary processors may then be forced into console I/O mode using targeted HALT -CPU commands.

4. Console primary to/from console secondary.

   The console primary must be able to send complete commands to a console secondary. This allows the primary to update the copy of an implementation-specific

parameter stored in each processor. Such commands are generated internally by the console program. Also, commands entered at the console terminal which are intended for a secondary must be forwarded by the primary.

Secondaries must be able to send complete messages to the primary. Such messages arrive complete, the primary can easily avoid interleaving messages on the console terminal.

2.6 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Integrated ECO #30
2. Widget –> Device or Controller, as appropriate
3. VMS –> OpenVMS
4. Converted appropriate internal text to various 'notes'
5. Convert to SDML

Revision 4.1, August 12, 1991

1. Replace previous Console Chapter with Console ECO #15
2. Includes 3 chapters and two appendices, renumber I/O Chapter
3. Material substantially changed or rearranged

\

Chapter 3
System Bootstrapping (IV)

This chapter describes the net effects of the action of the console to control the system platform hardware. The major system state transitions and the role of the console in controlling those transitions is described in Section 3.1.1. When power is applied to an Alpha system, the console initializes the system as given in Section 3.2. The console actions necessary to bootstrap system software are described in Section 3.3. These steps include processor initialization (Section 3.3.1.6), memory sizing and testing (Section 3.3.1.1), building an initial virtual address space (Section 3.3.1.3), and loading the bootstrap (Section 3.5). The console actions to restart system software are described in Section 3.4.

3.1 Processor States and Modes

3.1.1 States and State Transistions

An Alpha processor can be in one of five major states:

1. Powered off - no system power supplied to the processor.
2. Halted - operating system software execution suspended.
3. Bootstrapping - attempting to load and start the operating system software.
4. Restarting - attempting to restart the operating system software.
5. Running - operating system software functioning.

The transitions between the major states are determined by the current state and by a number of variables and events, including:

- Whether power is available to the system.
- The console AUTO_ACTION environment variable.
- The console lock setting.
- The Bootstrap-In-Progress (BIP) flags.
- The Restart-Capable (RC) flags.
- Processor error halts.
- The CALL_PAL HALT instruction.
- Console commands.

The following is a key for Figure 3–1:

A Console is unlocked and AUTO_ACTION is "HALT" (544C 4148_{16}).

B Console is unlocked and AUTO_ACTION is "BOOT" (544F $4F42_{16}$).

C Console is unlocked and AUTO_ACTION is "RESTART" (54 5241 5453 4552_{16}) or console is locked.

D Console is unlocked, the processor is forced into console I/O mode.

Figure 3–1: Major State Transitions

Action Causing Transition to Final State	Initial State					
	Off	Halted	Booting	Restart	Running	
Powerfail	Off	Off	Off	Off	Off	Final State
A and Power Restored	Halted					
B and Power Restored	Booting					
C and Power Restored	Restart					
BOOT and Console Is Locked		Booting				
START or CONTINUE or Console Is Unlocked		Running				
Bootstrap Fails or D			Halted			
Bootstrap Succeeds			Running			
D				Halted		
Restart Fails				Booting		
Restart Succeeds				Running		
A and Processor Halts or D					Halted	
B and Processor Halts					Booting	
C and Processor Halts					Restart	

To effect major state transitions, the console obeys these rules:

- If the console is unlocked when power is restored or when the processor halts, enter the state selected by the console AUTO_ACTION environment variable.
- If the console is locked when power is restored or when the processor halts, attempt a processor restart.
- When processor restart fails, attempt a bootstrap of that processor. One cause of a failed restart is the processor's RC flag being clear when the console attempts the restart.
- When system bootstrap fails, halt. One cause of a failed bootstrap is the processor's BIP flag being set prior to the console attempting the bootstrap. Only the processor that failed bootstrap will halt.
- When system bootstrap or processor restart succeeds, the processor starts running.
- When the primary processor is halted and the console is unlocked, the console BOOT command causes a system bootstrap.
- When a secondary processor is halted and the console is unlocked, the console START -CPU command causes the console to attempt to start that processor running.
- When a processor is halted and the console is unlocked, the console CONTINUE command cause the processor to continue running as though no halt was incurred.
- If the console is unlocked and a specified processor is running or booting or restarting, that processor is halted by a console HALT -CPU command.

IMPLEMENTATION NOTE

In an embedded console implementation, the primary processor must be forced into the console I/O mode prior to issuing the HALT -CPU command; see Section 3.7.3.

3.1.2 Major Modes

In addition to the major states, the console and processor are described as being in one of three modes:

1. Program I/O mode

   The processor is running. The processor interprets instructions, services interrupts and exceptions, and initiates I/O operations under the control of the operating system.

2. Console I/O mode

   The processor is halted or bootstrapping or restarting. The console provides control over the system; The operating system has either relinquished control

or has yet to gain control. The operating system does not service interrupts or exceptions or initiate I/O operations. The actions of the console are determined by internal console state and commands from the console operator.

3. Console Initialization mode

   The console has yet to acquire control of the processor. The console itself may also require initialization, such as when power is first applied to the system.

A given processor may be in one of four modes:

1. Primary processor in program I/O mode or "running primary"
2. Primary processor in console I/O mode or "console primary"
3. Secondary processor in program I/O mode or "running secondary"
4. Secondary processor in console I/O mode or "console secondary"

As noted in Section 1.1, implementations must include a mechanism to force a processor executing in program I/O mode into console I/O mode.

3.2 System Initialization

An Alpha system must be initialized when power is restored. System initialization also occurs as the result of a system bootstrap when the BOOT_RESET environment variable is set to "ON" ($4E4F_{16}$), or as the result of the console INITIALIZE command. Initialization involves all implementation-specific, system-wide actions necessary to give the system the ability to boot system software on the primary processor. Table 3–1 summarizes the effects of initialization as seen by system software.

Initialization may include initialization of the console itself. During console initialization, the console must build the HWRPB and all associated data structures necessary to permit the console to accept console commands and boot system software.

System initialization may also include any necessary system bus, processor, or I/O device initialization. The initialization of a processor performed as part of system initialization is not necessarily that performed just prior to transfer of control to the operating system bootstrap. See Section 3.3.1.6 for a description of processor initialization as seen by system software.

Table 3–1: Effects of Power-Up Initialization

Processor State	Initialized State:
BIP and RC flags	Cleared
Reason for halt code	'0' (bootstrap)
Integer and floating point registers	UNPREDICTABLE
System memory	Unaffected if preserved by battery backup; otherwise, UNPREDICTABLE

Table 3–1 (Cont.): Effects of Power-Up Initialization

Processor State	Initialized State:
Environment variables	Unaffected if non-volatile otherwise, set to default
BB_WATCH	Unaffected
I/O device registers	UNPREDICTABLE

3.3 System Bootstrapping

This section describes the operations performed by the Alpha console to locate, load, and transfer control to a primary bootstrap. The responsibilities of the console and the initial state seen by system software are presented for multiprocessor and the uniprocessor environments. The actions of the console for cold bootstrap (full hardware initialization) and warm bootstrap (partial hardware initialization) are described.

A system bootstrap can occur as the result of a powerfail recovery, a processor halt, or an INITIALIZE or BOOT console command. See Section 3.1.1 for a complete description of these state transitions.

3.3.1 Cold Bootstrapping in a Uniprocessor Environment

This section describes a cold bootstrap in a uniprocessor environment. A system bootstrap will be a cold bootstrap when any of the following occur:

- Power is first applied to the system
- A console INITIALIZE command is issued and the AUTO_ACTION Environment variable is set to "BOOT" ($544F\ 4F42_{16}$).
- The BOOT_RESET environment variable is set to "ON" ($4E4F_{16}$).
- Requested by system software.

The console must perform the following steps in the cold bootstrap sequence.

1. Perform a system initialization
2. Size memory
3. Test sufficient memory for bootstrapping
4. Load PALcode
5. Build a valid Hardware Restart Parameter Block (HWRPB)
6. Build a valid Memory Data Descriptor Table in the HWRPB
7. Initialize bootstrap page tables and map initial regions
8. Locate and load the system software primary bootstrap image

9. Initialize processor state on all processors

10. Transfer control to the system software primary bootstrap image

The steps leading up to the transfer of control to system software may be performed in any order. The final state seen by system software is defined, but the implementation-specific sequence of these steps is not. Prior to beginning a bootstrap, the console must clear any internally pended restarts to any processor.

3.3.1.1 Memory Sizing and Testing

Memory sizing is the responsibility of the console. The console must also test sufficient memory to permit control to be passed to the primary bootstrap image. The results of console memory sizing and testing are passed to system software in the Memory Data Descriptor (MEMDSC) Table located by HWRPB[200].

The MEMDSC Table contains one or more memory cluster descriptors. Each memory cluster descriptor describes a physically contiguous extent of physical memory within which there are no holes. Cluster descriptors are ordered by increasing physical address; the range of PFNs described by cluster N is of lower address than the range of PFNs described by cluster N+1.

The MEMDSC Table must be quadword aligned and both physically and virtually contiguous. The MEMDSC Table format is shown in Figure 3–2; the memory cluster descriptor format is shown in Figure 3–3. The size of the MEMDSC Table can be determined by the number of clusters contained in MEMDSC[16]. The size of the table and the offset to the last quadword of the table are given by:

```
MEMDSC_SIZE = ((7 * MEMDSC[10₁₆]) + 3) * 8
MEMDSC_END  = MEMDSC_SIZE -8
```

The memory within a cluster is either available to system software or reserved for console use. Usage within a cluster cannot be mixed; if the cluster contains a page reserved for console use, system software cannot allocate any page within the cluster. The memory cluster descriptor contains a cluster usage field which indicates the cluster availability to system software. Note that the primary bootstrap image must reside in clusters available to system software.

The memory within each cluster may be fully tested, partially tested, or untested by the console. If the memory is untested, no cluster memory bitmap is built. The console must test enough memory to allow the primary bootstrap image to be loaded and control to be passed to that image. This memory includes:

- PALcode memory and scratch areas
- CPU logout areas
- Memory bitmaps
- HWRPB and all offset blocks
- Console CRB map entries
- Bootstrap address space page tables
- Primary bootstrap image

- One page for the initial bootstrap stack

Any additional memory testing by the console is implementation-specific. It is the responsibility of system software to test any memory untested by the console.

A cluster bitmap is built if the cluster is available to system software and the console tests any memory within the cluster. Each page in the cluster is represented by a bit in the bitmask. A '1' in the bitmap means that the corresponding page is "good"; the page was tested without error. A '0' in the bitmap means that the corresponding page is "bad"; the page is either untested or was tested but encountered correctable (Corrected Read Data) errors or hard (Read Data Substitute) errors.

Cluster bitmaps must be at least quadword aligned and must be an integral number of quadwords; any unused bits in the highest addressed quadword MBZ.

\See Section 3.7.1 for the rationale behind memory clusters, highwater marking, and marking Corrected Read Data errors as bad pages.\

Figure 3–2: Memory Cluster Descriptor Table

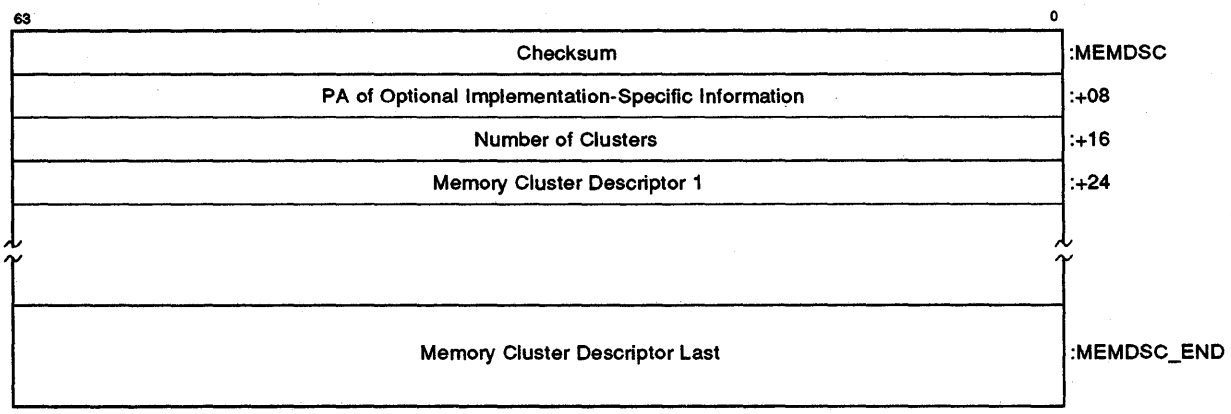


Table 3–2: Memory Cluster Descriptor Table Fields

Offset	Description
MEMDSC	CHECKSUM - Checksum which is the 64-bit, 2's complement sum ignoring overflows of all the quadwords from MEMDSC+8 through MEMDSC_END. The checksum does not include any of the cluster bitmaps nor any optional implementation-specific data.
+08	IMP_DATA_PA - Physical address of additional implementation-specific information (if any). If no additional implementation-specific information exists, the field must contain a zero.
+16	CLUSTERS - Number of clusters in the Memory Cluster Descriptor Table. Unsigned integer.

Table 3–2 (Cont.): Memory Cluster Descriptor Table Fields

Offset	Description
+24	CLUSTER - Each Memory Cluster Descriptor describes an extent of physical memory. See Figure 3–3.

Figure 3–3: Memory Cluster Descriptor

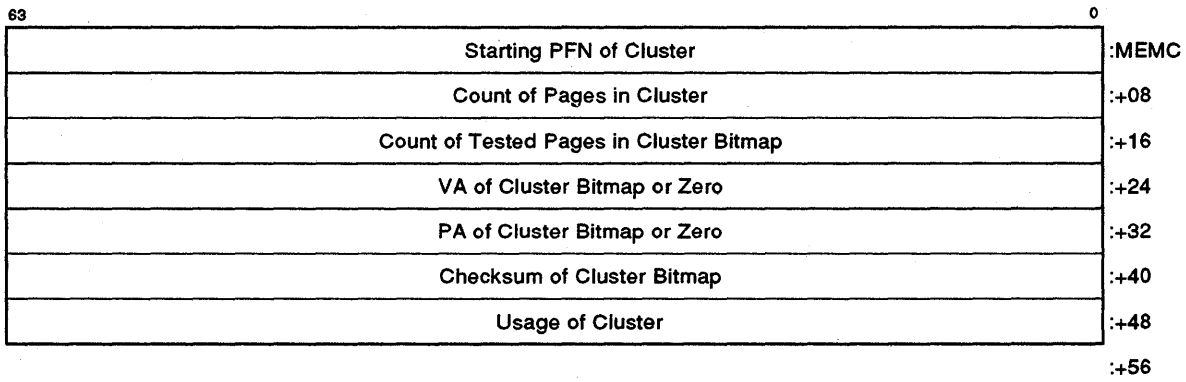


Table 3–3: Memory Cluster Descriptor Fields

Offset	Description
MEMC	PFN - Starting PFN of the memory cluster.
+08	PAGES - Number of pages in the memory cluster. Unsigned integer.
+16	TESTED_PAGES - Number of tested memory pages in the cluster. If only a limited extent of the cluster memory was tested, a bitmap is built, and this field indicates the number of pages that were tested. See Section 3.7.1.
+24	BITMAP_VA - Starting virtual address of the cluster memory testing bitmap in the bootstrap address space. If the memory is untested, no bitmap is built and this field is set to zero.
+32	BITMAP_PA - Starting physical address of the cluster memory testing bitmap. If the memory is untested, no bitmap is built and this field is set to zero.
+40	BITMAP_CHECKSUM - Checksum which is the 64-bit, 2's complement sum ignoring overflows of the cluster memory testing bitmap. Computed over the PAGES active bits only.
+48	USAGE - Indicates whether the cluster is available for use by system software. If USAGE<0> is '0', system software may allocate and use the cluster. If USAGE<0> is '1', the cluster is reserved for console use and must not be allocated by system software. USAGE<63:1> SBZ.

3.3.1.2 PALcode Loading

The console loads PALcode into good memory within a memory cluster which is not available to system software. If PALcode scratch space is required, the console allocates good memory within a memory cluster which is not available to system software. PALcode memory and scratch space are at least page aligned. The console records the starting physical address and length of PALcode memory and scratch space and then sets the PALcode Memory Valid (PMV) flag in the per-CPU slot of the primary processor. The PMV flag indicates that the PALcode descriptors are valid.

After PALcode loading and initialization, the console sets the PALcode Loaded (PL) and PALcode Valid (PV) flags in the primary's per-CPU slot. The PL flag indicates that PALcode has been loaded; the PV flag indicates that any necessary PALcode initialization has been performed.

PALcode loading and initialization is implementation-specific. The PALcode source may be a special console device, ROM, a system device, a communications line, or any other implementation-specific source. The state of the console and system must be such that the source is accessible. The means by which any PALcode internal state is initialized is implementation-specific.

3.3.1.3 Bootstrap Address Space

\ See Section 3.7.5 for a justification of the structure of the initial bootstrap address space.\

All system software, including the primary bootstrap image, runs in a virtual memory environment. The console creates the initial page tables which define the initial bootstrap address space for the primary bootstrap. System software may replace this bootstrap address space at any time after the console passes control to the primary bootstrap image.

The bootstrap address space consists of four regions. All regions must be located in good memory within clusters which are available to system software. The regions are:

Region 0

This region maps all console or PALcode data structures which must be shared with system software. These structures include the HWRPB in its entirety, all blocks located by HWRPB offsets, the console callback routines, and all memory bitmaps. Region 0 begins at address 256MB, virtual address 0000 0000 1000 0000_{16}. The starting address of the HWRPB is the base of Region 0.

Region 1

The primary bootstrap image is loaded into this region. The region must be at least large enough to load system software plus three pages. The three additional pages are used as an initial bootstrap stack and stack guard pages. The stack guard pages are virtually adjacent to the bootstrap stack page and marked no-access. All other pages in the region are mapped and valid. Region 1 begins at address 512MB, virtual address 0000 0000 2000 0000_{16}.

SOFTWARE NOTE

This region must be set to the size of the primary bootstrap image plus 3 pages for OpenVMS Alpha and at least 256K bytes for OSF/1 Alpha.

Region 2

This region, or "page table space", contains the bootstrap address space page tables. Region 2 begins at address 1GB, virtual address 0000 0000 4000 0000_{16}. The range is dependent on the page size:

Page Size	Page Table Space Address Range
8KB	1GB to 1GB+8MB
16KB	1GB to 1GB+16MB
32KB	1GB to 1GB+32MB
64KB	1GB to 1GB+64MB

This region includes the level 2 and level 3 page tables used to map all three regions comprising bootstrap address space. The level 2 page table maps itself as a level 3 page table. The address of the level 2 page table page and the PTE within the page which is used for self-mapping are also dependent on the page size:

Page Size	Virtual Address of Level 2 Page Table	L2PTE Number Used for Self-mapping
8KB	1GB+1MB	128
16KB	1GB+512KB	32
32KB	1GB+256KB	8
64KB	1GB+128KB	2

Figure 3–5 illustrates the initial page tables that map the virtual address regions shown in Figure 3–4.

Region 3

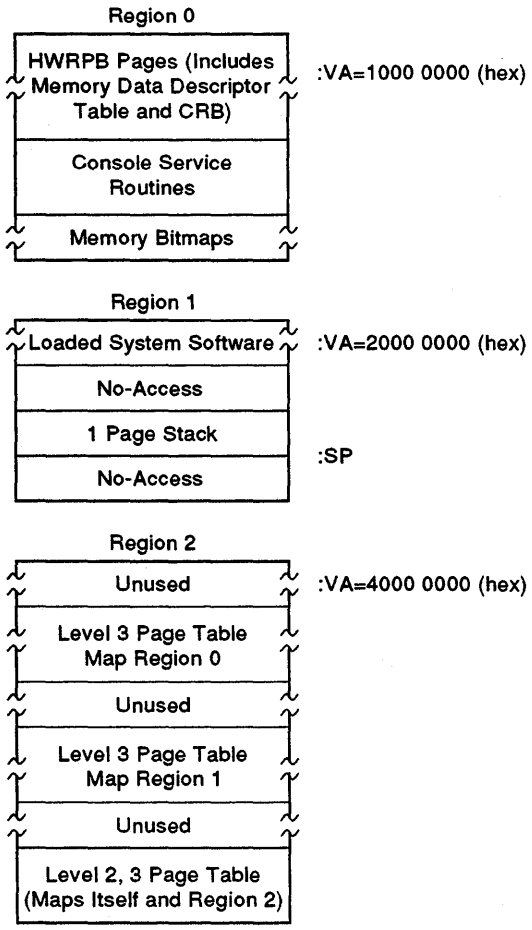
This region maps the level 1 page table pages. The level 1 page table is self-mapped by the penultimate PTE in the page. Region 3 exists to support virtual page table lookup for Translation Buffer misses. Region 3 is not the primary page table space that is presented to bootstrap software; system software must explicitly map the level 1 page tape page if required.

PROGRAMMING NOTE

Due to the self mapping, Region 3 maps all page table pages. The level 2 and level 3 page table pages are in both Region 2 and Region 3.

Page Size	Virtual Address of Level 1 Page Table
8KB	2**64–8GB–8MB–16KB
16KB	2**64–64GB–32MB–32KB
32KB	2**64–.5TB–128MB–64KB
64KB	2**64–4TB–.5GB–128KB

Figure 3–4: Initial Virtual Memory Regions

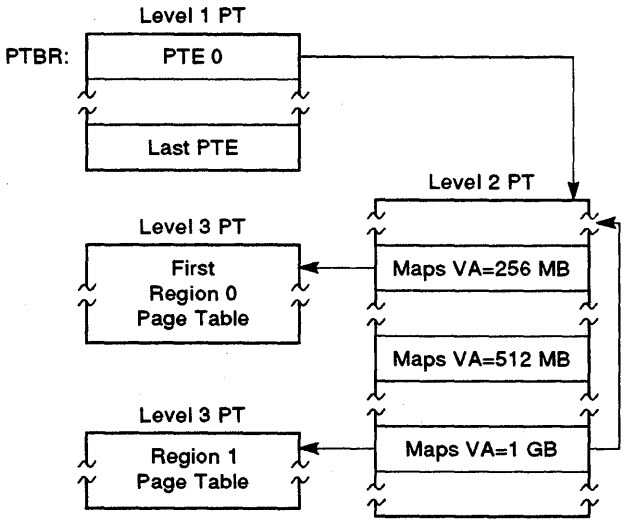


All valid pages allow read/write access from kernel mode and deny all access from executive, supervisor and user modes. All fault bits (FOR, FOW, FOE) are clear, as well as Address Space Match (ASM) and Granularity Hint (GH).

The self-mapping of the level 2 page table excludes the level 1 page table page from Region 2. The level 1 page table has two active PTEs. The first L1PTE points to

the PFN of the level 2 page table page which maps page table space (Region 2). The penultimate L1PTE contains the PFN of the level 1 page table itself, thus defining Region 3. Only these two entries within the level 1 page table are valid; all other level 1 PTEs are zeroes. See Section 3.7.5.

Figure 3–5: Initial Page Tables

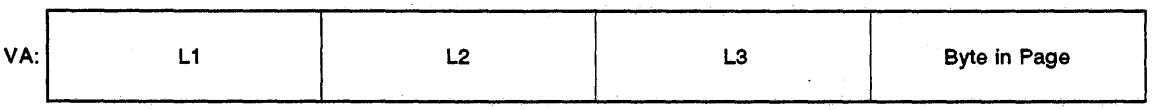


The level 2 PT maps Region 2 (page table space) at 1 GB. The level 2 PT maps itself as its own level 3 PT.

The level 1 PT is not mapped.

The self-mapping of the level 2 page table also causes the addresses of the level 2 and level 3 PTEs for a given virtual address to be functions of that address. For every virtual address within the bootstrap address space, there is exactly one location within page table space for the level 2 PTE that maps that virtual address, and exactly one location for the level 3 PTE that maps that virtual address.

Thus, the level 2 and level 3 PTE virtual addresses for a given virtual address (VA) within bootstrap address space can be calculated given the page size. The following bit range definitions provide convenient notation for referring to the constituent parts of a virtual address. For example, "VA<L2>" is equivalent to "VA<32:23>" for 8KB sized pages.



Page Size	L1	L2	L3
8KB	42:33	32:23	22:13
16KB	46:36	35:25	24:14
32KB	50:39	38:27	26:15
64KB	54:42	41:29	28:16

The base of page table space is a constant value:

1. `PT_Base = 1GB`

   The virtual address of the level 3 PTE (L3PTE_VA) of any virtual address (VA) is given by:

2. `L3PTE_VA(VA) = PT_Base + (page_size * VA<L2>) + (8 * VA<L3>)`

   Thus, the virtual address of the level 3 PTE which maps the lowest address of page table space is given by:

   ```
   L3PTE_VA(PT_Base) = PT_Base + (page_size * PT_Base<L2>)
   ```

   Since the level 2 page table is self-mapped, the above is also the base virtual address of the level 2 page table. Thus:

3. `L2PT_Base = PT_Base + (page_size * PT_Base<L2>)`

   Finally, the virtual address of the level 2 PTE (L2PTE_VA) of any virtual address (VA) is given by:

   ```
   L2PTE_VA(VA) = L2PT_Base + (8 * VA<L2>)
   ```

4. `L2PTE_VA(VA) = PT_Base + (page_size * PT_Base<L2>) + (8 * VA<L2>)`

3.3.1.4 Bootstrap Flags

The Bootstrap-In-Progress (BIP) and Restart-Capable (RC) processor state flags in the primary processor's per-CPU slot are used to detect failed bootstraps. If the primary reenters console I/O mode while the BIP flag is set and the RC flag is clear, the bootstrap attempt fails, and the subsequent console action is determined by Figure 3–1.

The console sets the BIP flag and clears the RC flag prior to transferring control to system software. System software sets the RC flag to indicate that sufficient context has been established to handle a restart attempt. System software clears the BIP flag to indicate that the bootstrap operation has been completed. The RC flag should be set prior to clearing the BIP flag.

Table 3–4: Console Interpretation of BIP and RC flags

BIP	RC	Interpretation at Entry to Console I/O Mode
set	clear	Failed bootstrap
set	set	Halt condition encountered during bootstrap, restart processor
clear	clear	Failed restart
clear	set	Halt condition encountered, restart processor

3.3.1.5 Loading of System Software

The console is responsible for loading system software at the base of Region 1 beginning at virtual address 512MB. This software is expected to be a primary bootstrap program which is responsible for loading other system software, but may be diagnostic or other special purpose software. Section 3.5 contains descriptions of the format of each supported bootstrap medium.

The console uses the BOOT_DEV environment variable to determine the bootstrap device and the path to that device. These environment variables contain lists of bootstrap devices and paths; each list element specifies the complete path to a given bootstrap device. If multiple elements are specified, the console attempts to load a bootstrap image from each in turn.

The console uses the BOOTDEF_DEV, BOOT_DEV, and BOOTED_DEV environment variables as follows:

1. At console initialization, the console sets the BOOTDEF_DEV and BOOT_DEV environment variables to be equivalent. The format of these environment variables is a function of the console implementation and independent of the console presentation layer; the value may be interpreted and modified by system software. See *Appendix E* for a list of current formats.
2. When a bootstrap results from a BOOT command which specifies a bootstrap device list, the console uses the list specified with the command. The console modifies BOOT_DEV to contain the specified device list. NOTE: This may require conversion from the presentationlayer format to the registered format.
3. When a bootstrap is the result of a BOOT command which does not specify a bootstrap device list, the console uses the bootstrap device list contained in the BOOTDEF_DEV environment variable. The console copies the value of BOOTDEF_DEV to BOOT_DEV.
4. When a bootstrap is not the result of a BOOT command, the console uses the bootstrap device list contained in the BOOT_DEV environment variable. The console does not modify the contents of BOOT_DEV.
5. The console attempts to load a bootstrap image from each element of the bootstrap device list. If the list is exhausted prior to successfully transferring control to system software, the bootstrap attempt fails and the subsequent console action is determined by Figure 3–1.

6. The console indicates the actual bootstrap path and device used in the BOOTED_DEV environment variable. The console sets BOOTED_DEV after loading the primary bootstrap image and prior to transferring control to system software. The BOOTED_DEV format follows that of a BOOT_DEV list element.

7. If the bootstrap device list is empty, BOOTDEF_DEV or BOOT_DEV are NULL 00_{16}, the action is implementation-specific. The console may remain in console I/O mode or attempt to locate a bootstrap device in an implementation-specific manner.

The BOOT_FILE and BOOT_OSFLAGS environment variables are used as default values for the bootstrap filename and option flags. The console indicates the actual bootstrap image filename (if any) and option flags for the current bootstrap attempt in the BOOTED_FILE and BOOTED_OSFLAGS and environment variables. The BOOT_FILE default bootstrap image filename is used whenever the bootstrap requires a filename and either none was specified on the BOOT command or the bootstrap was initiated by the console as the result of a major state transition. The console never interprets the bootstrap option flags, but simply passes them between the console presentation layer and system software.

3.3.1.6 Processor Initialization

Before control is transferred to system software, certain IPRs and other processor state must be initialized as shown in Table 3–5. Processor initialization is performed by the console prior to booting a processor, prior to restarting a processor, or as the result of the INITIALIZE -CPU console command.

The Context Valid (CV) flag in the processor's per-CPU slot must be valid for processor initialization to be successful. If the CV flag is clear, the HWPCB contained in the per-CPU slot is not valid, and the console must not transfer control to system software. In the event of this or any error initializing the processor, the console retains control of the system and generates the binary error message ERROR_PROC_INIT.

Table 3–5: Processor Initialization

Processor State		Initialized State
ASN	Address Space Number	Zero
ASTEN	AST Enable	ASTEN in processor's HWPCB
ASTSR	AST Summary	ASTSR in processor's HWPCB
FEN	Floating Enable	FEN in processor's HWPCB
IPL	Interrupt Priority Level	31
MCES	Machine Check Error Summary	Zero
PCBB	Privileged Context Block	Address of processor's HWPCB
PS	Processor Status	IPL=31, VMM=0, CM=K, SW=0

Table 3–5 (Cont.): Processor Initialization

Processor State		Initialized State
PTBR	Page Table Base Register	PFN value in processor's HWPCB
SISR	Software Interrupt Summary	Zero
WHAMI	Who-Am-I	CPU identifier
SCC	System Cycle Counter	Zero
SP	Kernel Stack Pointer	KSP in processor's HWPCB
Other IPRs		UNPREDICTABLE
Cache, instruction buffer, or write buffer		empty or valid
Translation buffer		Invalidated
Main memory		Unaffected
Integer and floating point registers		Unaffected, except SP
Reason for Halt code		Unaffected
BIP and RC flags		Unaffected
Environment variables		Unaffected

3.3.1.7 Transfer of Control to System Software

Prior to transferring control to system software, the console must define valid hardware privileged context for that software. The console builds that context in the hardware privileged context block (HWPCB) in the primary processor's per-CPU slot. The initialize context is summarized in Table 3–6.

The initial KSP points to the lowest addressed quadword in the higher addressed stack guard page (top-of-stack) of Region 1 of the bootstrap address space. The PTBR points to the level 1 page table page. All other scalar and floating point register contents are UNPREDICTABLE.

After building HWPCB for the primary, the console sets the Context Valid (CV) flag in the primary's per-CPU slot. All other bootstrap information is passed from the console to system software via environment variables. See Section 2.2 for more details.

Table 3–6: Initial HWPCB contents

HWPCB Field	Initialized State
KSP	Top-of-stack (contents of SP)
ESP	UNPREDICTABLE
SSP	UNPREDICTABLE
USP	UNPREDICTABLE

Table 3–6 (Cont.): Initial HWPCB contents

HWPCB Field	Initialized State
PTBR	PFN of level 1 page table
ASN	Zero
ASTSR	Zero
ASTEN	Zero (all disabled)
FEN	Zero (disabled)
PCC	Zero
Unique Value	Zero
PAL scratch	Implementation-specific

Control is transferred to system software in kernel mode at IPL 31 with virtual memory management enabled. Control is transferred to the first longword of the system software image loaded into Region 1, virtual address 0000 0000 2000 0000_{16}. Prior to transferring control, the console ensures that the SP contains the KSP value in the HWPCB. System software should assume that the stack is initially empty.

The transfer of control transitions the primary processor from the halted state into the running state and from console I/O mode into program I/O mode. The rest of the uniprocessor bootstrap process is the responsibility of system software.

3.3.2 Warm Bootstrapping in a Uniprocessor Environment

The actions of the console on a warm bootstrap are a subset of those for a cold bootstrap. A system bootstrap will be a warm bootstrap whenever the BOOT_RESET environment variable is set to "OFF" (46 $4E4F_{16}$) and console internal state permits.

The console performs the following steps in the warm bootstrap sequence.

1. Locate and validate the Hardware Restart Parameter Block (HWRPB)
2. Locate and load the system software primary bootstrap image
3. Initialize processor state on all processors
4. Initialize bootstrap page tables and map initial regions
5. Transfer control to the system software primary bootstrap image

At warm bootstrap, the console does not load PALcode, does not modify the Memory Data Descriptor Table, and does not reinitialize any environment variables. If the console cannot locate and validate the previously initialized HWRPB, the console must initiate a cold bootstrap. Prior to beginning a bootstrap, the console must clear any internally pended restarts to any processor.

PROGRAMMING NOTE

Warm bootstrap permits system software to preserve limited context across bootstraps. See Sections 2.5.2 and 3.7.1.

3.3.2.1 HWRPB Location and Validation

After console initialization, the console must preserve the location of the HWRPB in an implementation-specific manner. On warm bootstraps and restarts, the console locates the HWRPB and verifies it by ensuring that:

1. The first quadword of the table contains the physical address of the table.
2. The second quadword of the table contains "HWRPB" 0000 0042 5052 5748_{16}.
3. The quadword at offset HWRPB[288] contains the 64-bit, 2's complement sum ignoring overflows of the quadwords from offset HWRPB[00] to HWRPB[280], inclusive, relative to the beginning of the potential HWRPB.
4. The quadword at offset [0] of the MEMDSC block contains the 64-bit, 2's complement sum ignoring overflows of the quadwords from MEMDSC+8 through MEMDSC_END of that block. The MEMDSC block is located by the MEMDSC OFFSET at HWRPB[200]. See Figure 3–2.
5. As described in Section 2.1.4, if a CONFIG table exists, it is located by the CONFIG OFFSET at HWRPB[208]. The quadword at offset [8] of the optional CONFIG table contains the 64-bit, 2's complement sum ignoring overflows of the quadwords from CONFIG+8 through CONFIG_END of that table.

If any of the above conditions are not true, the HWRPB is not valid. The warm bootstrap (or restart) fails. The subsequent console action is determined by Figure 3–1. If a bootstrap is indicated, a cold bootstrap will be performed.

\The console must not search memory for a HWRPB; searching memory constitutes a security hole.\

3.3.3 Multiprocessor Bootstrapping

Multiprocessor bootstrapping differs from uniprocessor bootstrapping primarily in areas relating to synchronization between processors. In a shared memory system, processors cannot independently load and start system software; bootstrapping is controlled by the primary processor.

3.3.3.1 Selection of Primary Processor

The primary processor is selected by the console during system initialization prior to any access to main memory by any processor. Selection of the primary processor may be done in any fashion that guarantees choosing exactly one primary processor.

Once a primary processor has been selected, the secondary processors take no further action until appropriately notified by the primary processor. In particular, secondary processors must not access main memory.

See Section 3.7.3 for considerations for embedded console implementations.

3.3.3.2 Actions of Console

After selection, the console proceeds to bootstrap the primary processor following the normal uniprocessor bootstrap as described in Section 3.3.1.

The console must correctly initialize all HWRPB fields used for synchronization or communication between the processors. The console must initialize the PRIMARY CPU ID field at HWRPB[32], zero the TXRDY and RXRDY bitmasks at HWRPB[296] and HWRPB[304], and recompute the HWRPB checksum at HWRPB[288].

The console must also initialize each per-CPU slot for the secondary processors. The console must:

1. Clear the BIP, RC, OH, and CV flags.
2. Clear the Halt Request code field.
3. Set the PP flag if the processor is present.
4. Set the PA flag if the processor is present and available for use by system software.
5. Set the PMV and PL flags if the console has loaded PALcode on this processor.
6. Set the PV flag if the console has initialized PALcode on this processor.
7. Set the PE processor variation flag if the processor is eligible to become a primary.

After initializing each processor's per-CPU slot, the console must notify each console secondary processor of the existence and location of the valid HWRPB. See Section 3.7.3 for considerations for embedded console implementations.

3.3.3.3 PALcode Loading on Secondary Processors

Most console implementations load PALcode on all secondary processors prior to bootstrapping the primary processor. Console implementations may delay the loading or initialization of PALcode on a secondary. If delayed, PALcode loading and initialization requires the cooperation of system software executing on the running primary and the console executing on behalf of the secondary.

The console secondary must have performed any necessary initialization as described in Section 3.3.3.5. All interprocessor console communications follow the mechanisms described in Section 2.4. The operation proceeds as follows:

1. The console secondary initializes the PALcode memory and scratch space length fields in its per-CPU slot.
2. The console secondary sets the PALcode major revision, minor revision, and compatibility subfields in the PALcode revision field in its per-CPU slot.
3. The console secondary notifies the primary that PALcode loading is requested by transmitting a ?PALREQ? message to the running primary as described in Section 2.4.
4. The console secondary polls the PALcode Memory Valid (PMV) flag in its per-CPU slot.

5. The running primary detects the console secondary request.
6. The running primary verifies that the Processor Available (PA) flag is set in the secondary's per-CPU slot. If not, the operation fails.
7. The running primary compares the major and minor revision sub-fields of the PALcode revision field in its per-CPU slot to that in the secondary's per-CPU slot. If the revisions levels do not match, the running primary proceeds to step 12.
8. The running primary compares the number of processors currently sharing its PALcode image to the maximum contained in the sub-field of the PALcode revision field of its per-CPU slot. If the current number is the maximum, no additional console secondary can share the PALcode image. The running primary proceeds to step 12.

PROGRAMMING NOTE

The running primary can determine the number of processors currently sharing a given PALcode image by counting the number of per-CPU slots with the same valid PALcode memory space descriptors. A PALcode memory space descriptor is valid if the PALcode Loaded (PL) flag is set in the per-CPU slot.

9. The running primary copies the PALcode memory and scratch space descriptors from its per-CPU slot into the secondary's per-CPU slot.
10. The running primary copies the PALcode variation, compatibility, and maximum number of processors sub-fields of the PALcode revision field from its per-CPU slot into the secondary's per-CPU slot.
11. The running primary sets the PALcode Loaded (PL) flag in the secondary's per-CPU slot, then proceeds to step 13.
12. The running primary allocates physical memory for PALcode memory and scratch areas and records the addresses in the secondary's per-CPU slot.
13. The running primary sets the PALcode Memory Valid (PMV) flag in the secondary's per-CPU slot.
14. The console secondary observes the PMV flag is set in its per-CPU slot.
15. If the PL flag in its per-CPU slot is not set, the console secondary loads PALcode into the allocated PALcode memory and scratch space. In this case, the console secondary sets the PALcode Loaded (PL) flag in its per-CPU slot.
16. The console secondary ensures that any required implementation-specific PALcode initialization is performed.
17. The console secondary sets the PALcode Valid (PV) flag in the secondary's per-CPU slot.

The PALcode memory and scratch space must be page aligned. If not allocated by the console prior to system bootstrap, the allocation management of PALcode memory for secondary processors is the responsibility of system software.

Note that it is the responsibility of system software to ensure that the PALcode revision levels of all processors are compatible. This may be performed by the primary prior to starting the secondary, by the starting secondary, or any combination thereof. PALcode images of different revision levels are compatible if the PALcode revision compatibility subfields match.

3.3.3.4 Actions of the Running Primary

System software executing on the primary processor must initialize the HWPCB for each secondary processor. The HWPCB contains the necessary privileged context for the execution of system software and successful restarts. The HWPCB must be initialized prior to requesting that the console secondary perform any START command. After initializing the HWPCB, system software sets the Context Valid (CV) flag.

Once the PALcode is valid on a console secondary, the secondary waits for a START (or other) command from the running primary. System software issues the necessary console commands which instruct the secondary to begin executing software. The exchange of commands and messages between the running primary and a secondary is described in Section 2.4.

PROGRAMMING NOTE

Note that all commands sent to a console secondary are implicitly targeted to the secondary. No -CPU command qualifier is necessary.

3.3.3.5 Actions of a Console Secondary

After failing to become the primary, a console secondary uses an implementation-specific mechanism to determine when a valid HWRPB has been constructed in main memory. The console secondary then locates the HWRPB in an implementation-specific manner.

Once the HWRPB is located, the secondary locates its per-CPU slot using its CPU ID as an index. The secondary verifies that its slot exists by comparing its CPU ID to the number of per-CPU slots at HWRPB[144]. If its CPU ID exceeds the number of per-CPU slots, the secondary must not leave console mode or continue to access main memory. If PALcode loading is necessary, the console secondary follows the procedure given in Section 3.3.3.3.

Once PALcode is valid, the console secondary waits for a START (or other) command from the running primary by polling the appropriate flag in the RXRDY bitmask. The exchange of commands and messages between the running primary and a secondary is described in Section 2.4.

In response to a START command, the console secondary:

1. Verifies that the Context Valid (CV) flag is set in its per-CPU slot.

2. Sets the Bootstrap-In-Progress (BIP) flag in its per-CPU slot.
3. Clears the Restart-Capable (RC) flag in its per-CPU slot.
4. Initializes the processor.
5. Loads the privileged context specified by the HWPCB in its per-CPU slot.
6. Loads the procedure value at HWRPB[264] into R27.
7. Clears R26 (return address) and R25 (argument information).
8. Loads the virtual address page table base (VPTB) register with the value stored in HWRPB[272].
9. Transfers control to the CPU Restart routine, whose virtual address is stored in HWRPB[256].

The CV flag indicates that the HWPCB in the slot contains valid hardware privileged state for system software. If the CV flag is not set, the processor remains in console I/O mode. '

3.3.3.6 Bootstrap Flags

The Bootstrap-In-Progress (BIP) and Restart-Capable (RC) processor state flags in the console secondary processor's per-CPU slot are used to control error recovery during secondary starts. If the secondary reenters console I/O mode while the BIP flag is set and the RC flag is clear, the start attempt fails. Failed starts are equivalent to failed bootstraps, and the subsequent console action is determined by Table Figure 3–1. See Section 3.3.1.4 and Table 3–4.

3.3.4 Addition of a Processor to a Running System

A processor may be added to a running system at any time if a slot has been provided for it in the HWRPB. The new console secondary processor follows the secondary start procedure given in Sections 3.3.3.3 and 3.3.3.5 with one minor difference. If no PALcode loading is necessary, the console secondary sends a ?STARTREQ? message to the running primary. This message notifies the primary that a new processor has been added to the configuration. After sending the ?STARTREQ? message, the console secondary waits for a START (or other) command from the running primary. See Section 2.4 for a description of interprocessor console communication.

3.3.5 System Software Requested Bootstraps

System software can request that the console perform a system bootstrap. This request can be made on any processor in a multiprocessor system and overrides the setting of the AUTO_ACTION and BOOT_RESET environment variables.

To request a bootstrap, system software sets one of the bootstrap requested codes in the Halt Request field of its per-CPU slot then executes a CALL_PAL HALT instruction. If a cold bootstrap is requested, the "Cold Bootstrap Requested" code ('2') is set; the "Warm Bootstrap Requested" ('3') code is set to request a warm bootstrap.

Rather than the normal error halt processing described in Section 3.4.4, the console initiates the appropriate system bootstrap as described in Sections 3.3.1 and 3.3.2.

The bootstrap attempt is unconditional; the AUTO_ACTION or the BOOT_RESET environment variables do not affect the bootstrap attempt.

3.4 System Restarts

The console is responsible for restarting a processor halted by powerfail or by error halt. The console follows the same sequence for a primary or secondary processor.

3.4.1 Actions of Console

The console begins the restart sequence by locating and then validating the HWRPB following the procedure given in Section 3.3.2.1. If the HWRPB is not valid, the restart attempt fails. See Section 3.1.1 for console actions at major state transitions.

If the HWRPB is valid, the console uses the processor CPU ID as an index to calculate the address of that processor's HWRPB slot. The console:

1. Verifies that the processor's PALcode Valid (PV) flag is set. If the PV flag is clear, PALcode is not valid, and restart attempt fails.
2. Verifies that the processor's Context Valid (CV) flag is set. If the CV flag is clear, the HWPCB does not contain valid software context for the restart, and the restart attempt fails.
3. Examines the processor's restart-capable (RC) flag. If set, the console proceeds with the restart at step 5. If clear, system software is not capable of attempting the restart, the restart attempt fails.
4. Examines the Bootstrap-In-Progress (BIP) flag. If clear, and the AUTO_ACTION environment variable is "BOOT" (544F 4F42$_{16}$), a system bootstrap is attempted. Otherwise, the processor remains in console I/O mode. See Figure 3–1.
5. Loads the privileged context specified by the HWPCB in its per-CPU slot.
6. Loads the procedure value at HWRPB[264] into R27.
7. Clears R26 (return address) and R25 (argument information).
8. Loads the virtual address page table base (VPTB) register with the value stored in HWRPB[272].
9. Transfers control to the CPU Restart routine, whose virtual address is stored in HWRPB[256].

On all restart attempt failures the console initiates the action indicated by Figure 3–1. Note that the PV and CV flags should never be clear for the primary processor; if either flag is clear, then the restart fails. Also note that no PALcode or system software is loaded during a restart.

It is the responsibility of system software to complete the restart operation and to set the RC flag at the point where a subsequent restart can be handled correctly.

3.4.2 Powerfail and Recovery - Uniprocessor

An Alpha system requires power to operate. The system power supply conditions external power and transforms it for use by the processor, memory, and I/O subsystems. Backup options are available on some systems to supply power after external power fails. The backup option may supply power to all of the system platform hardware, or only a subset.

The effect of an external power failure depends on the backup option.

1. If no backup option exists, the processor is not restartable after restoration of power. The processor must be bootstrapped or left halted in console I/O mode.
2. If the backup option maintains power to all of the system platform hardware, execution of system software is unaffected by the power failure. It must be possible for system software to determine that a transition to backup power has occurred.
3. If the backup option maintains only the contents of memory and keeps system time with the BB_WATCH, the power supply must request a powerfail interrupt. After requesting the interrupt, the power supply must continue to supply power to the processor for an implementation-specific period to allow system software to save state.

In the last case, powerfail recovery is possible only if adequate system state is preserved during an interruption of power to the processor. As explained in *OpenVMS Section, Chapter 6*, a powerfail interrupt is delivered at IPL 30 to the interrupt service routine located at SCB offset 640_{16}. System software must save all volatile state and perform any operating system specific actions necessary to ensure later successful recovery.

When power is restored, the console determines that the HWRPB is still valid, then examines the console lock and AUTO_ACTION environment variable. If the console is locked, and AUTO_ACTION environment variable is "RESTART" (54 5241 5453 4552_{16}), the console attempts an operating system restart. See Section 3.1.1.

Note that the processor may lose state when power is lost. For example, if a processor is halted when power fails, the action on power up is still determined by the console switches and environment variables. The system does not necessarily stay halted.

3.4.3 Powerfail and Recovery - Multiprocessor

There are two basic approaches to powerfail recovery on multiprocessor systems:

- United - all available processors effectively experience the powerfail event identically.
- Split - each available processor effectively experience independent powerfail events.

A processor is "available" if the Processor Available (PA) flag is set in the processor's per-CPU slot. The Powerfail system variation flag at HWRPB[88] indicates the type of powerfail and restart action.

A multiprocessor Alpha system that supports powerfail recovery must implement the united powerfail mode. The split mode may be optionally implemented as a alternative selected at system bootstrap.

SOFTWARE NOTE

OpenVMS Alpha supports only the united powerfail and recovery mode at this time. Powerfail recovery is possible only when the primary is restarted; all secondaries should remain in console I/O mode.

3.4.3.1 United Powerfail and Recovery

In united powerfail and recovery mode, all available processors experience powerfail interrupts, halts, and restorations uniformly. If one available processor experiences a powerfail event, all other available processors experience that event. Therefore, if one processor powerfails and recovers, all processors must do so. Even if a separately powered processor does not actually lose power, that processor will still receive the powerfail interrupt and must be restarted as if power had been lost.

When power is restored and a restart is to be attempted, the console must determine whether to restart all available processors or only the primary processor. The console determines the appropriate action by the Powerfail Restart (PR) flag in the system variation field of the HWRPB[88]. If the PR flag is set, the console attempts to restart all available processors; if clear, the console attempts to restart only the primary processor. In both cases, it is the responsibility of system software to coordinate and synchronize further powerfail recovery.

3.4.3.2 Split Powerfail and Recovery

In split powerfail and recovery mode, only the available processors that actually experience a loss of power will see a powerfail interrupt and subsequent recovery. Available processors that are separately powered and do not lose power do not see a powerfail interrupt.

When power is restored and a restart is to be attempted, the console must determine whether to restart any available processor or only the primary processor. As in the united mode, the console determines the appropriate action by the Powerfail Restart (PR) flag in the system variation field of the HWRPB[88]. If the PR flag is set, the console attempts to restart any available processor. If clear, the console attempts to restart only the primary processor; on a secondary, the console sends the ?STARTREQ? message and waits for a START (or other command) from the running primary as discussed in Section 3.3.3.5. Again, system software has the responsibility for further coordination and synchronization of powerfail recovery.

3.4.4 Error Halt and Recovery

There are a number of serious error conditions that prevent a processor from executing the current thread of software. Such error conditions are detected by PALcode and lead to the processor being halted.

The console must ensure that the processor hardware state when a halt is encountered is visible to system software after a subsequent restart attempt and to the console operator. This state includes the current values in PS, PC, SP, PCBB, HWPCB, all integer registers, all floating point registers, and the name of the halt condition. The console must:

1. Ensure that the contents of the integer and floating point registers appear unaffected.
2. Write the current hardware context to the HWPCB located by the current PCBB.
3. Write the current PS, PC, PCBB register contents into the processor's per-CPU slot.
4. Write the current R25, R26, and R27 register contents into the processor's per-CPU slot.
5. Set appropriate code into the Reason For Halt field of the processor's per-CPU slot.

Note that the values of R25, R26, and R27 must be explicitly saved in the per-CPU slot to permit the console to follow the Alpha calling standard when invoking the CPU Restart routine.

Section 3.1.1 and Table 2–3 list the defined halt conditions that transition an Alpha processor from the running state to a halted state, and which may lead to an attempt to restart the processor. Each condition is passed to the operating system in the Reason For Halt quadword of the processor's HWRPB slot.

When an error halt occurs, the console examines the console lock setting. If the console is locked, the console attempts a restart. If unlocked, the console action is determined by the setting of the AUTO_ACTION environment variable, see Figure 3–1. See Section 3.4.1 for a description of the restart attempt process.

The processor must be initialized after an error halt. If the processor starts running after an error halt without an intervening processor initialization, the operation of the processor is UNDEFINED. The effects of processor initialization are summarized in Table 3–5.

An error halt directly affects only the processor that incurred one, although multiple processors may simultaneously and coincidentally incur their own error halt conditions. If restarts are enabled, each halted processor must be independently restarted by the console. The restarts of individual processors may occur in a different order than the error halts occurred, but if the console restarts any halted processor, it must restart all halted processors in a timely fashion unless a bootstrap is requested in the meantime. A bootstrap nullifies any pending restarts in the multiprocessor.

3.4.5 Operator Requested Crash

When the operating system does not respond to normal program requests, the console operator may request that the console request an operating system crash. A console requested crash differs from a console halt of a processor in that system software can write a crash dump.

The console operator interacts with the console presentation layer and requests the crash with a HALT -CRASH command. The console converts this command to an error halt restart of system software. After gaining control of the processor, the console preserves the hardware state; see Section 3.4.4. The console passes the crash request to system software by using the "Console Operator requests system crash" code in the Reason For Halt field in the primary's per-CPU slot. It is the responsibility of the system software restart routine to initiate the crash in an implementation-specific fashion.

3.4.6 Primary Switching

System software may find it necessary to replace the primary processor with one of the running secondary processors without bootstrapping the system. This "switch" of the running primary may be caused by an error encountered by the primary, or by a program request. Switching a running primary must be initiated by system software; the console cannot force a switch to occur.

Support for primary switching is optional to system software, console implementations, and system platforms. The system platform hardware must permit the selected secondary to assume the functions of a primary. The selected secondary must have direct access to the console, a BB_WATCH, and all I/O devices. Direct access to the console ensures that the secondary can access console I/O devices and the console terminal. Direct access to a BB_WATCH ensures that the secondary can act as the system timekeeper. Direct access to all I/O devices ensures that the secondary can initiate I/O requests to and receive I/O interrupts from all I/O devices, and that the secondary can reinitialize all devices as part of powerfail recovery.

If the processor is eligible to become a primary, the console will set the Primary Eligible (PE) processor variation flag in the processor's per-CPU slot during processor initialization.

Primary switching requires cooperation between system software and the console. System software is responsible for the selection of the new primary and any necessary redirection of I/O interrupts. The console is responsible for any necessary configuration of the console terminal or other console device interface.

The sequence of events differs depending on the type of console implementation. On a system with an embedded console, the operation proceeds as follows:

1. System software performs any actions specific to system software synchronization.
2. System software executing on the old primary ensures that the console terminal is in a quiescent state. In particular, character reception from the terminal must be suspended.
3. System software selects the new primary. The selected secondary must be eligible as indicated by the PE processor variation flag in its per-CPU slot.
4. System software executing on the old primary invokes the PSWITCH console callback specifying the "transition from primary" action.

5. The console attempts to perform any necessary hardware state changes to transform the old primary into a secondary.

HARDWARE/SOFTWARE COORDINATION NOTE

An example of such a hardware state change is disabling a console UART physically located on the processor board.

6. If the state change is completed, PSWITCH returns success status. System software may proceed with the primary switch at step 8.
7. If the state change is not effected, PSWITCH returns failure status. System software must take other appropriate action.
8. System software executing on the old primary notifies system software on the selected secondary of the successful PSWITCH completion.
9. System software executing on the selected secondary invokes the PSWITCH console callback specifying the "transition to primary" action.
10. The console verifies that the selected secondary is eligible to become a primary and attempts to perform any necessary hardware state changes to transform the old secondary into the new primary.\An example of such a hardware state change is draining the character FIFO and enabling a console UART physically located on the processor board.\
11. If the state change is completed, PSWITCH returns success status. System software may proceed with the primary switch at step 13.
12. If the state change is not effected, PSWITCH returns failure status. System software must select a different potential primary or take other appropriate action.
13. System software executing on the selected secondary reactivates the console terminal. In particular, character reception from the terminal is reenabled.
14. System software performs any additional system reconfiguration, updates the PRIMARY CPU ID field at HWRPB[32], recomputes the HWRPB checksum at HWRPB[288], and performs any actions specific to system software synchronization.

On a system with a detached console, the operation is similar, but only one call to PSWITCH is required. Additional calls to PSWITCH with the "switch primary" action may result in UNDEFINED operation. The operation proceeds as follows:

1. System software performs any actions specific to system software synchronization.
2. System software executing on the old primary ensures that that the console terminal is in a quiescent state. In particular, character reception from the terminal must be suspended.
3. System software selects the new primary. The selected secondary must be eligible as indicated by the PE processor variation flag in its per-CPU slot.

4. System software executing on any processor invokes the PSWITCH console callback specifying the "switch primary" action and the CPU ID of the new primary.
5. The console verifies that the selected secondary is eligible to become a primary and attempts to perform any necessary hardware state changes to transform the old primary into a secondary and to transform the selected secondary into the primary.
6. If the state change is completed, PSWITCH returns success status. System software may proceed with the primary switch at step 9.
7. If the state change is not effected and the resulting hardware state permits a return to system software, PSWITCH returns failure status. System software must select a different potential primary or take other appropriate action.
8. If the state change is not effected and the resulting hardware state does not permit a return to system software, the console takes the action associated with a failed restart.
9. System software executing on the selected secondary reactivates the console terminal. In particular, character reception from the terminal is reenabled.
10. System software performs any additional system reconfiguration, updates the PRIMARY CPU ID field at HWRPB[32], recomputes the HWRPB checksum at HWRPB[288], and performs any actions specific to system software synchronization.

3.4.7 Saving and Restoring console terminal state during HALT/RESTART

Abrupt transitions from program I/O mode to console I/O mode may occur. Such transitions may be caused by execution of a CALL_PAL HALT instruction, a catastrophic error, or a console operator forcing the processor into console I/O mode. Upon transition to console I/O mode, the console must be able to regain control of the console terminal, even though system software may have changed the device characteristics.

The console may seize control of the console terminal without regard to system software when the transition is such that no return to program I/O mode is possible. Such transitions are normally associated with a catastrophic error.

If system software execution may be continued, the console must be able to restore the existing state of the console terminal. The console must regain and subsequently relinquish control of the console terminal with the cooperation of system software.

HARDWARE/SOFTWARE COORDINATION NOTE

This is particularly desirable on workstations when the console operator forces the processor into console I/O mode.

System software may provide SAVE_TERM and RESTORE_TERM routines which can be called by the console to save and restore the state of the console terminal. To provide these optional routines, system software loads the SAVE_TERM and RESTORE_TERM starting virtual address and procedure descriptor fields in

the HWRPB, and recomputes the HWRPB checksum at HWRPB[288]. At system bootstraps, the console sets these fields to zero.

The console calls SAVE_TERM and RESTORE_TERM in kernel mode at IPL 31 in the memory management policy established by system software. The console loads the routine procedure value into R27, clears R25 and R26, and then transfers control to system software at the starting virtual address. The procedure value and starting virtual address for SAVE_TERM are contained in HWRPB[224] and [232]; those for RESTORE_TERM are contained in HWRPB[240] and [248]. These routines are invoked only on the primary processor and only upon an unexpected entry into console I/O mode. Note that the console must preserve sufficient hardware state to permit the processor to be restarted prior to invoking these routines. See Section 3.4.4.

Exit from these routines must be accomplished by using the CALL_PAL HALT instruction to return the processor to console I/O mode; these routines do not use the RET subroutine return instruction. Prior to exit, these routines must set the "SAVE_TERM/RESTORE_TERM exit" code ('1') in the Halt Request field of the primary's per-CPU slot and indicate success ('0') or failure ('1') status in R0<63>. The console will not attempt to continue system software in the event that a failure status is returned.

SAVE_TERM and RESTORE_TERM may be called when system software has encountered an unexpected CALL_PAL HALT or other halt condition; system state may be corrupt. These routines must be written with little or no dependencies on possibly corrupt system state.

HARDWARE/SOFTWARE COORDINATION NOTE

A console terminal on a serial line may or may not have state which needs to be saved. A console terminal on a workstation may require the system software to "roll down" the current screen to expose the "console window" and "roll up" the "console window" to expose the current screen.

3.4.7.1 SAVE_TERM - Save Console Terminal State

Format:

```
status     = SAVE_TERM
```

Inputs:

None

Outputs:

status = R0; status:

R0<63>	'0'	Success, terminal state saved.
	'1'	Failure, terminal state not saved.
R0<62:0>	SBZ	

SAVE_TERM is called by the console after an unexpected entry to console mode. The routine performs any implementation-specific and device-specific actions necessary to save the state of the console terminal as established by system software. When the routine exits and console I/O mode is restored, the console is free to modify the existing console terminal state in any manner.

3.4.7.2 RESTORE_TERM - Restore Console Terminal State

Format:

```
status    = RESTORE_TERM
```

Inputs:

None

Outputs:

status = R0; Status:

R0<63>	'0'	Success, terminal state restored
	'1'	Failure, terminal state not restored
R0<62:0>	SBZ	

RESTORE_TERM routine is called by the console just prior to continuing system software. The routine performs any implementation-specific and device-specific actions necessary to restore the state of the console terminal as established by system software.

3.4.8 Operator Forced Entry to Console I/O Mode

The console operator can force a processor into console I/O mode with a HALT -CPU command. When a processor enters console I/O mode in this way, the console sets the Operator Halted (OH) flag in its per-CPU slot. The console does not update the Reason For Halt or any other processor halt state in its per-CPU slot. The console sets the OH flag only as the result of an explicit operator action; the OH flag is not set on transitions to console I/O mode resulting from error halt conditions, powerfails, CALL_PAL HALT instructions in kernel mode, console operator requests of a system crash, or software directed processor shutdowns.

The console clears the OH flag prior to returning to program I/O mode as the result of a CONTINUE or BOOT command. The console may clear OH flag if an error halt or operator-induced condition is encountered which precludes a subsequent CONTINUE command. Such a condition is treated as an error halt; see Section 3.4.4.

3.5 Bootstrap Loading and Image Media Format

An Alpha console may load a primary bootstrap image from one or more of the device classes listed in Table 3–7. \A given console implementation may support any combination of the devices and protocols below; see Section 3.7.6.\ Subsequent sections describe how the console locates, sizes, and loads the bootstrap image for each device class.

Table 3–7: Bootstrap Devices and Image Media

Device Class	Data Link	Protocol
Local Disk	n/a	-Bootblock
Local Tape	n/a	-ANSI -Bootblock
Network-like	NI, FDDI	-MOP -Bootp -Bootparam -SNMP -CMIP
ROM	n/a	-ROM Bootblock
Console Storage	n/a	-Bootblock -Implementation-specific
Serial	DDCMP	-MOP

As explained in Section 3.3.1.5, the console attempts to load a bootstrap image from each element of a bootstrap device list until a successful image load is achieved. If the bootstrap image cannot be located or if the load fails for any reason, the console retains control of the system, generates the binary error message AUDIT_BSTRAP_ABORT, and then attempts to load a bootstrap image from the next bootstrap device list element. After a bootstrap image is successfully located and loaded, the console transfers control to system software as described in Section 3.3.

As the bootstrap image load proceeds, the console optionally generates an audit trail of messages indicating progress. The ENABLE_AUDIT environment variable controls audit trail generation. The audit trail begins with the AUDIT_BOOT_STARTS message. The audit trail continues with messages which are specific to the bootstrap device. All message codes generated by the console are summarized in Table Table 1–1; each consists of a binary message code which is interpreted by the console presentation layer.

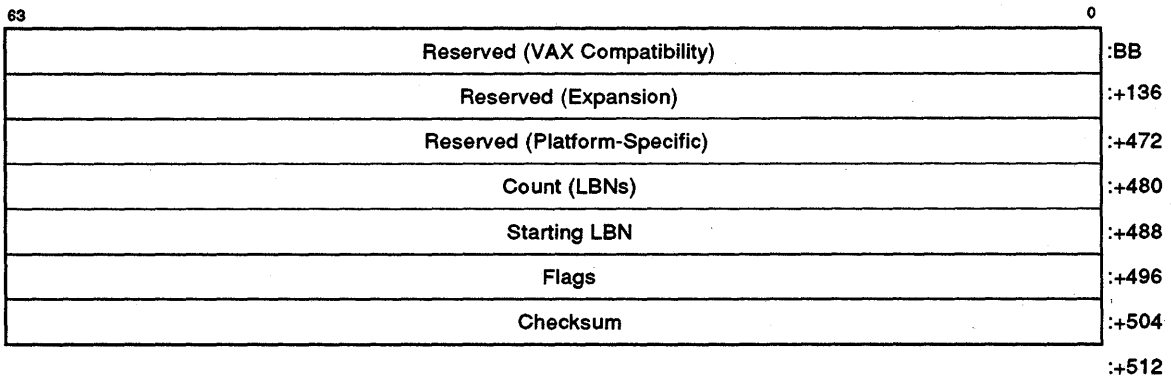
3.5.1 Disk Bootstrapping

An Alpha primary bootstrap may be loaded from a directly accessed disk device. The console loads the "boot block" contained in the first logical block (LBN 0) of the disk. The boot block contains the starting logical block number (LBN) of the primary bootstrap program and the count of contiguous LBNs which make up that image.

The first 512 bytes of the boot block are structured as shown in Figure 3–6. The console loads the primary bootstrap without knowledge of the operating system file system. The boot block is (previously) initialized by the operating system. The actual size of a logical block is device-specific and may exceed 512 bytes. The platform-specific quadword is unused by the operating system.

One intended use of this quadword is to permit a given console to boot another console which presents a different operating system interface. This quadword is intended for use only on locally connected disks which are not served to multiple, possibly nonhomogeneous, platforms. Note that neither OpenVMS or OSF/1 support this quadword. In particular, the quadword is lost at disk initialization, not written as a part of bootstrap block update and not replicated on a backup or archive.

Figure 3–6: Alpha Boot Block



A local disk bootstrap proceeds as follows:

1. The console reads the boot block from LBN 0 of the specified disk device.
2. The console validates the boot block CHECKSUM; if the checksum is not validated, the bootstrap image load attempt aborts. The console computes the checksum of the first 63 quadwords in the block as a 64-bit, 2's complement sum ignoring overflow. Note that the computation includes both reserved regions. The computed checksum is compared to the CHECKSUM at [BB+504].
3. The console generates the AUDIT_CHECKSUM_GOOD message if the audit trail is enabled.
4. The console ensures that the FLAG quadword is zero; otherwise the bootstrap image load attempt aborts.
5. The console ensures that the COUNT is non-zero; otherwise the bootstrap image load attempt aborts. The count field indicates the number of contiguous logical blocks that contain the primary bootstrap.
6. The console generates the AUDIT_LOAD_BEGINS message if the audit trail is enabled.

7. The console reads the primary bootstrap image specified by COUNT and STARTING LBN into system memory; in the event of any error, the bootstrap image load attempt aborts.

   The transfer begins at the logical block given by the STARTING LBN; a contiguous COUNT number of logical blocks is read. The image is read into a virtually contiguous system memory buffer; the starting virtual address is 0000 0000 2000 0000_{16}. (See Section 3.3.1.3).

   Errors include device hardware errors, the specified STARTING LBN not being present on the disk, or unexpectedly encountering the last logical block on the disk during the read.

8. The console generates the AUDIT_LOAD_DONE message when the load has completed; the message is generated only if the audit trail is enabled.

9. The console prepares to transfer control to the bootstrap program as described in Section 3.3.1.7.

3.5.2 Tape Bootstrapping

An Alpha primary bootstrap may be loaded from a directly accessed tape device. Prior to loading the primary bootstrap, the console must determine the tape format and locate the primary bootstrap on the tape. The console:

1. Rewinds the tape on the specified tape device to the beginning of the tape (BOT).
2. Reads the first record.
3. Determines the record length.
   - If the record length is 80 bytes, the tape may be an ANSI-formatted tape. The console proceeds as described in Section 3.5.2.1.
   - If the record length is 512 bytes, the tape is "boot blocked". The console proceeds as described in Section 3.5.2.2.
   - If the length is other than 80 or 512 bytes, the bootstrap image load attempt aborts.

3.5.2.1 Bootstrapping From ANSI-formatted Tape

Prior to loading the primary bootstrap image from an ANSI-formatted tape, the console must ensure that the format is valid. To verify that a given record contains a particular ANSI label, the console checks for the ASCII label name string at the beginning of the record. For example, a record containing a VOL1 label begins with the ASCII string "VOL1". All other record bytes are ignored when verifying the label.

A primary bootstrap image filename may be specified explicitly on a BOOT command or implicitly by the BOOT_FILE environment variable. If no filename is specified, the first located file will be used.

A local ANSI-formatted tape bootstrap proceeds as follows:

1. The console verifies that the first record contains a VOL1 label; if the verification fails, the bootstrap image load attempt aborts.
2. The console generates the AUDIT_TAPE_ANSI message if the audit trail is enabled.
3. If no filename was specified, the console advances the tape position to the End-Of-Tape (EOT) side of the the first tape mark. The console proceeds to step 5.
4. If a filename was specified, the console attempts to locate that file on the tape. If the file cannot be located, the bootstrap image load attempt aborts. The console compares the specified filename with the filename present in each HDR1 label on the tape. At the first match, the console proceeds to step 5.

   The console searches for the specified file starting with the second tape record. The console reads 80-byte records from the tape until it encounters an HDR1 label, then proceeds as follows:

   a. The console generates the AUDIT_FILE_FOUND<filename> message, where <filename> is the value of the HDR1 label. The message is generated only if the audit trail is enabled.
   b. The console compares the specified filename with the 17 character File Identifier Field found in the HDR1 label.
   c. If a match occurs, then the console advances the tape position to after the next tape mark and proceeds to step 5. (Any HDR2 or HDR3 labels are ignored.)
   d. If there is no match, then the console advances the tape position over the next three tape marks and reads next the record. If another tape mark is found, then the logical end of volume has been encountered and the bootstrap image load attempt aborts. Otherwise the record should be the HDR1 label for the next file on the tape and the console proceeds at step a.

   The console aborts the bootstrap image load attempt whenever an unexpected tape mark is encountered, the tape runs off the end, or a hardware error occurs.
5. The console generates the AUDIT_LOAD_BEGINS message if the audit trail is enabled.
6. The console reads the primary bootstrap image from tape into system memory; in the event of any error or if the tape runs off the end, the bootstrap image load attempt aborts.

   The transfer from tape begins at the current tape position and continues until a tape mark is encountered. The image is read into a virtually contiguous system memory buffer; the starting virtual address is 0000 0000 2000 0000_{16}. (See Section 3.3.1.3).
7. The console checks that the bootstrap file was properly closed by:

   a. Reading the record after the tape mark and verifying that the record is an EOF1 label. If not, the bootstrap image load attempt aborts.

b. Searching for a subsequent tape mark. If one is not found, the bootstrap file was improperly closed and the bootstrap image load attempt aborts. (Any EOF2 and EOF3 labels are ignored.)

8. The console generates the AUDIT_LOAD_DONE message if the audit trail is enabled.

9. The console prepares to transfer control to the bootstrap as described in Section 3.3.1.7. Note that the console does not rewind or otherwise change the position of the tape after reading the bootstrap image.

3.5.2.2 Bootstrapping from Boot Blocked Tape

Bootstrapping from a boot blocked tape is similar to the local disk bootstrapping described in Section 3.5.1. The first tape record must be 512 bytes, and must follow the format given for disk boot blocks as shown in Figure 3–6. The STARTING LBN and FLAGS fields are MBZ for tape boot bootblocks.

All tape records which comprise the primary bootstrap must be 512 bytes in size. If the console encounters records of any other size, the bootstrap image load attempt aborts.

A local tape boot block bootstrap proceeds as follows:

1. The console generates the AUDIT_TAPE_BBLOCK message if the audit trail is enabled.

2. The console validates the boot block CHECKSUM; if the checksum is not validated, the bootstrap image load attempt aborts. The console computes the checksum of the first 63 quadwords in the block as a 64-bit, 2's complement sum ignoring overflow. Note that the computation includes both reserved regions and the MBZ fields. The computed checksum is compared to the CHECKSUM at [BB+504].

3. The console generates the AUDIT_CHECKSUM_GOOD message if the audit trail is enabled.

4. The console ensures that the COUNT is non-zero; otherwise the bootstrap image load attempt aborts. The count field indicates the number of subsequent 512 byte records that contain the primary bootstrap.

5. The console generates the AUDIT_LOAD_BEGINS message if the audit trail is enabled.

6. The console reads the COUNT subsequent records from the tape into system memory. The bootstrap image load attempt aborts if the console encounters any error, encounters any record size other than 512 bytes, or the tape runs off the end.

   The image is read into a virtually contiguous system memory buffer; the starting virtual address is 0000 0000 2000 0000_{16}. (See Section 3.3.1.3).

7. The console generates the AUDIT_LOAD_DONE message if the audit trail is enabled.

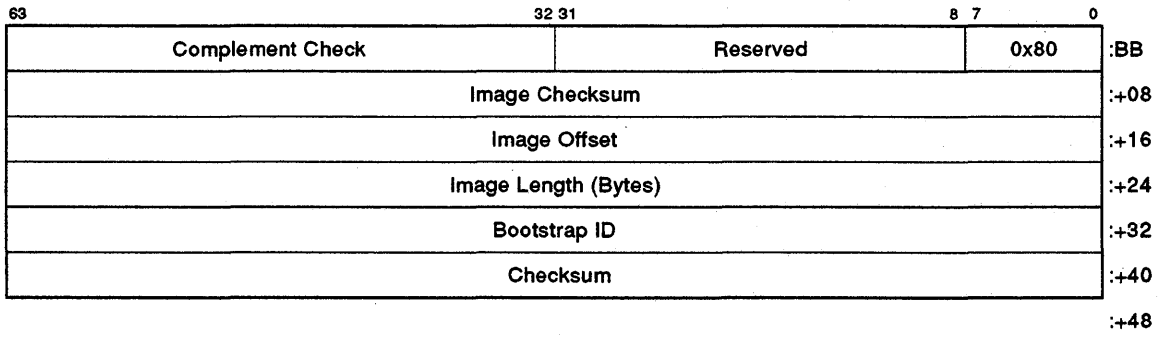
8. The console prepares to transfer control to the bootstrap as described in Section 3.3.1.7. Note that the console does not rewind or otherwise change the position of the tape after reading the bootstrap image.

3.5.3 ROM Bootstrapping

An Alpha console may support bootstrapping from Read Only Memory (ROM). Bootstrap ROM is assumed to appear in multiple discontiguous regions of the physical address space. A given ROM region may contain multiple bootstrap images. A given bootstrap image must not span ROM regions.

Each ROM bootstrap image is page aligned and begins with a boot block as shown in Figure 3–7. The ROM boot block is similar to the local disk and tape boot block shown in Figure 3–6.

Figure 3–7: Alpha ROM Boot block



A ROM bootstrap proceeds as follows:

1. The console locates the specified ordinal ROM bootstrap image; if the bootstrap image cannot be located, the bootstrap image load attempt aborts.

   The console locates the ROM bootstrap image by searching ROM regions beginning with the ROM region with the lowest physical address and proceeding upward to the ROM region with the highest physical address.

   The search proceeds as follows:

   a. The console verifies that the page contains a ROM bootstrap image:

      - The low-order byte of the first quadword must be 80_{16}.
      - The high-order longword of the first quadword must be the one's complement of the low-order longword.
      - The sixth quadword must contain the checksum of the first five quadwords. The checksum is computed as a 64-bit, 2's complement sum ignoring overflow.

b. The console generates the AUDIT_BOOT_TYPE<string> message for each valid bootblock, if the audit trail is enabled. The <string> is the ISO-LATIN-1 string contained in the BOOTSTRAP ID quadword.

c. If the specified ordinal image number has been reached, the console proceeds to step 2.

d. Otherwise, the console uses the IMAGE LENGTH at [BB+24] to determine the offset to the next ROM region page to be searched. The console repeats the process at step a.

2. The console computes the starting physical address of the bootstrap image by adding the physical address OFFSET at [BB+16] to the starting physical address of the bootblock [BB].

3. The console verifies the accessibility of each page of the bootstrap image. If any page is inaccessible, the bootstrap image load attempt is aborted.

4. The console generates the AUDIT_BSTRAP_ACCESSIBLE message if the audit trail is enabled.

5. If requested, the console validates the IMAGE CHECKSUM; if the checksum is not validated, the bootstrap image load attempt aborts. The console computes the checksum of all quadwords in the bootstrap image as a 64-bit, 2's complement sum ignoring overflow. The existence and implementation of the mechanism for requesting this validation is implementation-specific.

6. The console generates the AUDIT_BSTRAP_GOOD message if the audit trail is enabled.

7. If requested, the console copies the bootstrap image from ROM into system memory (RAM). The image is copied into a virtually contiguous buffer starting at virtual address 0000 0000 2000 0000_{16}. (See Section 3.3.1.3). The console generates the AUDIT_LOAD_BEGINS message before beginning the copy and the AUDIT_LOAD_DONE after the copy completes successfully if the audit trail is enabled.

8. The console prepares to transfer control to the bootstrap as described in Section 3.3.1.7.

3.5.4 Network Bootstrapping

An Alpha system may support bootstrapping over one or more network communication devices and data link protocols. The console actions are dependent on the network device, data link protocol, and remote server capabilities.

3.5.4.1 MOP-based Network Booting

An Alpha system can use the Digital Network Architecture Maintenance Operations Protocol to bootstrap an Alpha system; see the MOP specification for a detailed description.

The MOP bootstrap proceeds as follows:

1. The console determines if a bootstrap filename is to be used. The filename is taken from the BOOT command or the BOOT_FILE environment variable. If no

filename is specified on the BOOT command and BOOT_FILE is null, no filename will be used.

2. The console generates the AUDIT_BOOT_REQ<filename> message if the audit trail is enabled.
3. The console issues the appropriate MOP bootstrap request message(s).
4. The console receives an appropriate MOP response from a remote bootstrap server. If no such response is received, the bootstrap image load attempt aborts.
5. The console generates the AUDIT_BSERVER_FOUND message if the audit trail is enabled.
6. The bootstrap load proceeds following the MOP protocol.
7. When the console receives the first portion of the bootstrap image, the console generates the AUDIT_LOAD_BEGINS message if the audit trail is enabled.
8. The console loads the initial portion of the bootstrap image into a virtually contiguous system memory buffer; the starting virtual address is 0000 0000 2000 0000_{16}. (See Section 3.3.1.3).
9. When the bootstrap image has been loaded, the console generates the AUDIT_LOAD_DONE message if the audit trail is enabled.
10. The console prepares to transfer control to the bootstrap program as described in Section 3.3.1.7.

In the event of any error, the bootstrap image load attempt aborts.

3.5.4.2 BOOTP-UDP/IP Network Booting

TBD.

3.6 BB_WATCH

The following lists important points about BB_WATCH:

1. \ BB_WATCH is the correct name for this entity. Although incorrect terminology, T-O-Y, T-O-D-R, toy, todder, and watch chip when used in an Alpha context are equivalent in meaning to the BB_WATCH.\
2. System software must directly manipulate the BB_WATCH through an implementation-dependent interface.
3. System software makes the decision where to acquire known time; if a BB_WATCH is present, it may be used as the provider of known time.
4. Systems are not required to have a BB_WATCH.

SOFTWARE NOTE

However, all systems that support OpenVMS Alpha or OSF/1 on Alpha must have one.

5. If a BB_WATCH is present in a system, it meets the following requirements:
   - it has an accuracy of at least 50 ppm regardless of whether power is applied to the system;
   - it has a resolution of at least 1 second (That is, it is read and written in units of a second or better).
   - changing the entirety of the time maintained by the BB_WATCH takes under 1 second; and
   - it has battery backup to survive the loss of power.
6. A BB_WATCH is always accessible to the primary processor. Or stated another way, a processor must be able to access a BB_WATCH directly (i.e., not needing to go through another processor to get at it) in order to be a candidate for primary processor.
7. The number of BB_WATCHes in a system is either one for the entire system or one per each processor in the system; which of the two options a system chooses is implementation-dependent. If the latter option is chosen (one BB_WATCH per each processor), note that writing one BB_WATCH does not update another.
8. Although writing the BB_WATCH takes less than one second, it may not be a fast operation. Software should avoid frequently writing the BB_WATCH lest it negatively impact performance.
9. The processor and its PALcode never changes the value of BB_WATCH except under the direction of system software. (Note: the console, boot programs, and remote console clients are not system software.) The console, its PALcode, and any console application (including a diagnostic supervisor) never changes BB_WATCH except under the direction of the console operator – even when the CPU is HALTED, the processor is being initialized, or the BB_WATCH has an invalid time.

SOFTWARE NOTE

The format of time representation in the BB_WATCH may vary from implementation to implementation. The architecture requires, wherever possible, that when system software writes a time value into the BB_WATCH, the format of the time must conform to that of the 64-bit Absolute Time field of the 128-bit Digital Time Service Standard (DTSS) Binary Time field, as described in A-DG-ELEN112-00-0, Rev. A, 30-Jul-1987, which is available from Digital Standards and Methods Control. This absolute time format indicates the number of 100 nanosecond units that have elapsed since midnight October 15, 1582 UTC, the beginning of the Gregorian reform. Since the absolute time format is based on Coordinated Universal Time (UTC, popularly known as Green-

wich Mean Time or GMT), it does not include a local time offset.

If DTSS conformance is not possible in a particular BB_WATCH, system software must at least use UTC based time when writing the BB_WATCH.

This is a pure and simple constraint on the operating systems that use an Alpha system. It prevents an operating system from updating the BB_WATCH in an incompatible way with a subsequently booted operating system on the same machine.

This requirement is waivered for OpenVMS Alpha until no later than the first release of OpenVMS Alpha after 1-Jan-1995, when it will then comply.

PROGRAMMING NOTE

The Primary-Elegible (PE) bit in the per-cpu slot of the HWRPB for each processor indicates, among other things, whether the CPU has access to a BB_WATCH. See *Appendix D*.

The description of primary switching details the actions taken in a multiprocessor system, including the requirement for the primary processor to have access to the BB_WATCH.

3.7 Implementation Considerations

3.7.1 Memory Sizing, Testing, and Memory Data Descriptor Table

Alpha systems are allowed to have holes of unimplemented physical memory. The cluster mechanism allows all of available memory to be described in such a system without the need for creating bitmaps for unimplemented physical memory.

Every implementation cannot be required to test all of memory before booting the operating system. Partial memory testing is recommended whenever testing is time consuming and would significantly delay the bootstrapping process; the choice is implementation-specific. The highwater mark mechanism allows implementations to completely size memory without testing all of it and indicate to the operating system where testing ended.

This is the rationale for flagging pages that test as having Corrected Read Data errors as bad pages.

1. Pages which have hard (repeatable) or soft (transient) CRD errors must be reported as bad so that operating systems have the option of implementing a user-directed policy over the use of these pages. For example, OpenVMS Alpha customers with critical applications may want the operating system to use only pages that test absolutely good.

2. Determining whether a page that tests as a CRD page is really a CRD page or an RDS page is potentially a time consuming operation. A page of this type must have each bit held in a known state and all others put through a one to zero and zero to one transition to determine if the page is CRD or RDS. Flagging CRD pages as bad, frees the console from doing this extensive testing and potentially speeds-up the booting process. The operating system can bury this testing time with other tasks after it has been booted.
3. Typically the time between writing a test pattern to memory and reading it back is on the order of microseconds. The probability is low that a transient CRD error has occurred in this short time. Thus, pages testing as having CRD errors, probably have hard CRD errors and it is not efficient checking these CRD pages for the few times where the error is actually a transient error.

In some Alpha systems, it is expected that the console will attempt to partition physical memory into two clusters—one for the console and one for the operating system—and that all pages in the operating system cluster will be tested. Again, console implementations are strongly discouraged from testing all of memory if the booting process is significantly delayed.

Clusters reserved for console and PALcode use do not have associated bitmaps. If such a cluster would contain a large number (3 or more) of contiguous pages which encounter soft read errors or are otherwise unsuitable for console and PALcode, the console should consider breaking the bad pages into a separate cluster. This cluster should be made available for use by system software which can possibly reclaim the pages for use.

The PALcode function for flushing at least one page to memory (CFLUSH) may be used to aid in implementation of this system software function. (CFLUSH takes one argument, the PFN of the physical page to flush.)

The console does not alter the Memory Data Descriptor Table or any bitmaps across warm bootstraps. This permits system software to propagate information on system software memory testing and intermittent errors across operating system bootstraps. For example, system software could set the "bad" bit of a page which incurred repeated CRD errors.

3.7.2 Bootstrap Flags

The console uses the BIP and RC flags to detect failed bootstraps, starts, and restarts. The default response of the console is take the least drastic action possible. The console attempts a restart in preference to a bootstrap and attempts a bootstrap in preference to remaining in console I/O mode.

BIP and RC are shared between the console and system software. There are two improbable cases of seemingly extraneous bootstrap attempts:

1. Repeated power failures caused by a bouncing power supply.

   System software may not have sufficient time to set the RC flag.
2. Intermittent hardware failures on a secondary processor.

The console executing on the secondary may force a system bootstrap due to a failed restart of the secondary.

3.7.3 Embedded console

In an embedded console implementation, the console executes on the same processor as the operating system. In such an implementation, the state transitions as experienced by the processor are more conceptual. For example, the processor acting as the console will be executing instructions when in the halted state. The processor may also field console I/O mode exceptions and interrupts.

An embedded console may be implemented as an extension of PALcode or as a distinct software entity. The console may execute from dedicated RAM or ROM on the processor or, after console initialization, may execute from main memory.

An embedded console implementation must include a mechanism by which the primary processor can be forced into console I/O mode from program I/O mode. This enables the console operator to gain control of the system regardless of the state of the system software. See Section 1.2 for recommended and required mechanisms.

3.7.3.1 Multiprocessor considerations

In a multiprocessor system, selection of the primary processor occurs prior to any access to main memory by any of the processors. At system cold start, each of the processors will be executing in console I/O mode. The necessary memory for console execution must be independent of main memory; the console must be executing from dedicated console RAM or ROM and/or a suitably configured processor cache.

The selection of the console primary requires one or more hardware registers with state which is shared by all processors. One possible example is a mutex contained in a single-bit register accessed only with LDQ_L/STQ_C instructions. The primary successfully gains ownership of the mutex. Note that implementations should include mechanisms for operator override of the selection process and for recovery in the event that the selection process fails.

Once a console primary has been selected, the console secondaries take no further action until appropriately notified by the primary. In particular, console secondaries must not access main memory. The console primary has the responsibility of building the HWRPB and any console-internal data structures (such as environment variables) for the secondaries. When these structures have been initialized, the console primary must be able to signal one or more of the secondaries by additional hardware register(s).

The console primary allocates a HWRPB in main memory, initializes it, and stores its physical address in an implementation-specific non-volatile manner. The console primary then indicates the presence of the HWRPB and its location to all secondaries by an implementation-specific mechanism.

On system restarts, the console primary identifies itself by comparing its WHAMI register contents with the Primary CPU ID value stored in the HWRPB.

When executing in console I/O mode, all processors must observe the same values of all console environment variables. Of particular importance are the values of the

AUTO_ACTION and BOOT_RESET environment variables. After failing to become the console primary processor, a console secondary waits to be notified that a valid HWRPB exists. Upon such notification by the primary, the console secondaries use the address provided by the primary to locate the HWRPB. The primary may be in either program I/O mode or console I/O mode.

On cold bootstrap, a console secondary must not access main memory until notified by the primary that a valid HWRPB exists. Thus, there must exist a non-main memory based mechanism by which the primary may signal each of the secondaries. On warm bootstrap or restart, a secondary processor must locate its per-CPU slot in the HWRPB and poll its RXRDY bit.

Console processors must locate the HWRPB without searching memory; such a search constitutes a security hole. One possible implementation is to use an environment variable or other shared console data structure. The address of the HWRPB must be non-volatile across power failures in systems which support powerfail recovery.

Console implementations which support SAVE_ENV must be capable of executing the routine simultaneously on each processor. System software use of SAVE_ENV requires care. System software must invoke SAVE_ENV on all available processors, but cannot ensure that the non-volatile storage is updated on processors which are not available at the time of update. In the event of mismatch, the console uses the non-volatile values preserved by the primary processor.

3.7.4 Detached console

In a detached console implementation, the console executes on a separate and distinct hardware platform. A detached console may have cooperating special code which executes on one of the processors in the system configuration.

Detached console implementations should provide some sort of keep-alive function. System software should be able to detect failures of the path between the system platform and the console. This may be a single dedicated signal or may be periodic message exchange. System software should be capable of continuing to execute in the event of a keep-alive failure and restoration of the connection (or console state) should not cause a system crash or other major state transition. The console should buffer any messages in the event of a keep-alive failure until reconnection occurs.

Detached consoles may maintain a local console log. The logging device and format are implementation-specific.

3.7.5 Goals of the Bootstrap Address Space

The bootstrap address space established by the console for executing the primary bootstrap is specifically tailored to address the goals and needs of system software supported by Alpha, as listed here:

- The address space cannot exceed the reach of our supported implementation languages. In particular, page table address space must be reachable.
- The address space layout should not create conflicts for system software. The immediate addressing needs of system software must be accommodated.

- Page table simplicity is desirable, but not to the extent that bogus translation paths are created.
- The address space layout must be architected to ensure that the previous goals are met and to ensure a growth path for future console and primary bootstrap needs.
- The address space layout should not preclude bootstrapping an operating system which supports full 64-bit addressing.

Several alternatives were considered for implementing a bootstrap address space. One scheme that was considered involved having a single, triply-mapped page table. This scheme introduced address space conflicts which unnecessarily placed implementation restrictions on the primary bootstrap program. A variation of this which created additional page tables, all of which were naturally located in virtual memory, eliminated the bogus translation paths but didn't solve the address space conflicts created for the primary bootstrap program.

The chosen design solves all of these problems through careful location of page table address space. The location of page table address space naturally excludes the level 1 page table from virtual memory, but this is not a problem for software. The chosen design incurs no additional page table complexity or memory usage over any triply-mapped scheme that doesn't also introduce bogus translation paths.

3.7.5.1 Address Space must be reachable

There exists system software (OpenVMS Alpha Phase 1) which is implemented using 32-bit oriented languages. Such software is limited to a 32-bit address space subset modeled after the VAX address space and supported by Alpha longword arithmetic operations. This is not to say that the remainder of the Alpha virtual address space is inherently unavailable, only that the software implementation language imposes a restriction on the amount of the Alpha address space that can be reached by that particular software.

A requirement immediately emerges that the bootstrap address space in which system software executes must be "32-bit oriented". Valid potential bootstrap address space can only consist of the first and last 2GB (due to longword sign-extension) of the Alpha 64-bit virtual memory space.

3.7.5.2 The coarseness effect

A triply-mapped page table scheme of any kind imposes extreme coarseness upon the location of page table space. Consider Table 3–8, which shows the locations of page table space for a triply-mapped page table using different L1PTEs for self-mapping:

Table 3–8: Page Table Coarseness Effect

L1PTE Number	8KB	16KB	32KB	64KB
0	0	0	0	0

Table 3–8 (Cont.): Page Table Coarseness Effect

L1PTE Number	8KB	16KB	32KB	64KB
1	8GB	64GB	0.5TB	4TB
2	16GB	128GB	1.0TB	8TB
3	24GB	192GB	1.5TB	12TB
4	32GB	256GB	2.0TB	16TB
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Last	2**64–8GB	2**64–64GB	2**64–0.5TB	2**64–4TB

Self-mapping in any L1PTE other than the first L1PTE would locate page table space at an address that a 32-bit oriented language cannot reach.

3.7.5.3 Address Space must not create conflicts

3.7.5.3.1 Location of Page Table Space

As was noted above, the only reachable location for page table address space utilizes the first L1PTE, thus locating page table address space at a region beginning at address zero and extending at least to address 8GB-1. This creates an immediate addressing conflict since no reachable address space is left over for system software itself or for console-mapped structures and code.

A finer grained virtual address layout is therefore required, one in which the self-mapping that establishes reachable page table address space is done at page table level 2 instead of level 1. A level 1 page table would exist which is entirely empty except for the first L1PTE. The first L1PTE would point to a separate level 2 page table. A PTE within the level 2 page table would be used for self-mapping, thus locating page table address space at a finer grained location (within the total address space mapped by the single L1PTE) than would be otherwise possible. With this approach, page table space could be located within the entire 64-bit address space as shown in Table 3–9.

Table 3–9: Page Table Space Location

	Page Size			
L1PTE / L2PTE Numbers	8KB	16KB	32KB	64KB
0 / 0	0	0	0	0
0 / 1	8MB	32MB	128MB	0.5GB
0 / 2	16MB	64MB	256MB	1.0GB

Table 3–9 (Cont.): Page Table Space Location

L1PTE / L2PTE Numbers	Page Size 8KB	16KB	32KB	64KB
0 / 3	24MB	96MB	384MB	1.5GB
0 / 4	32MB	128MB	512MB	2.0GB
.	.	.	.	.

This table shows that self-mapping using any of the first four L2PTEs will define a reachable location for page table address space (anywhere in the first 2GB), regardless of page size.

Table 3–10 shows the size of page table address space as a function of page size. Any space in the first 2GB of virtual memory that is not part of page table address space is available for other uses.

Table 3–10: Page Table Address Space as Function of Page Size

Page Size	Length of Page Table Space
8KB	8MB
16KB	32MB
32KB	128MB
64KB	512MB

Self-mapping at level 2 naturally excludes the L1 page table from the defined page table address space. Self-mapping at level 2 merely establishes an address space within the context of whatever L1PTE is used to map the level 2 page table.

Either the level 1 page table can be mapped to some arbitrary, yet architected, VA *outside* of page table address space, or it can be left unmapped by the console, or another address space can be created through self-mapping at level 1 which would naturally include the level 1 page table. The need to support virtual PTE lookup during Translation Buffer miss processing dictates the third choice.

Thus the second L1PTE is used to map the level 1 page table itself. Note from the discussions above, this creates a second address sapce for the page tables which is not reachable from 32-bit oriented software. Such software will use the finer grained page table space created by the self-map technique at level 2.

3.7.5.3.2 Laying out the first 2GB

Bootstrap address space can be laid out once a location is chosen for page table space. The four natural locations for page table space that would be expressible regardless of page size are found in the column above for the 64KB page size. These locations are 0, 0.5GB, 1.0GB and 1.5GB. After reserving location zero for software use, any one of the remaining three locations could be chosen. Remaining address space in

the first 2GB could then be allocated for other purposes. The final layout of the first 2GB of address space is described in Section 3.3.1.3.

3.7.5.4 Conclusion

The needs of more restrictive implementation languages can be met by utilizing the natural flexibility of the Alpha multi-level page tables. This can be done without undue complexity or memory usage, and without precluding the use of any less restrictive language used to implement a '64-bit' operating system.

3.7.6 Bootstrap Devices and Image Media

Various factors should be considered when determining which of the bootstrap devices and image media listed in Table 3–7 are supported.

1. Workstations and other low-end platforms may consider supporting ROM bootblocks for DEC OSF/1 and customer applications. OpenVMS Alpha currently uses a single bootstrap image for all platforms; support for ROM bootstrapping will require customization. See Section 3.5.3 for the Alpha ROM Bootblock mechanism.
2. Platforms considering bootstrap media which is local to the console must negotiate with the operating systems for such support on a case-by-case basis. DEC OSF/1 supports the bootblock method; see Section 3.5.1.
3. Products intended for embedded systems applications should consider DDCMP /MOP support.

Support for audit trail generation during console bootstrap is strongly recommended to all implementations. An audit trail is essential to the isolation of errors during the bootstrap process. Section 3.5 give the architected audit trail for each bootstrap device. Console implementations may generate additional audit trail messages.

3.7.6.1 Disk Bootstrapping

Note that unlike the VAX boot block support, NO code is contained in the boot block; the boot block contains ONLY the LBN descriptor for the Alpha primary bootstrap image. Also note that an Alpha boot block can contains pointers to primary bootstrap images for both VAX and Alpha simultaneously.

Because the boot block includes an LBN and block count, the console need have no knowledge of the operating system file system or on-disk structure.

The first 136 bytes of the boot block are currently used by the VAX disk boot block mechanism. The next 80 bytes are not currently used either by VAX or Alpha boot blocks. For future expansions, VAX boot blocks should expand towards higher addresses, and Alpha boot blocks expand towards lower addresses; each region remains contiguous. These 216 bytes are ignored by the Alpha console except for the purposes of computing the bootblock checksum.

The boot block FLAGS word is reserved for future expansion. Flag<0> is reserved to indicate a discontiguous bootstrap image; Flag <63:1> are reserved for future definition. There are no current plans by any Digital operating system to have a discontiguous primary bootstrap image.

3.7.6.2 ROM Bootstrapping

A ROM block is uniquely identified as containing an Alpha bootstrap image by the value of 0080_{16} in the first word of the block. Each ROM bootstrap image is uniquely identified by a zero-based ordinal number.

The size of the ROM bootstrap is specified in bytes to permit the same ROM bootstrap image to be used by systems with different page sizes.

Other alternatives to specify which ROM bootstrap to use were considered, such as making the console operator give the physical address of the ROM bootstrap. The specified method seemed the least complicated. A console implementation should consider a command which permits the location of ROM bootstraps, their IDs, and PR assignments to be displayed.

Note that the specified searching process ensures that incorrectly created ROM bootblocks are ignored by the console.

3.7.6.3 Network Bootstrapping

Data link protocols include CSMA/CD (IEEE 802.3 and Ethernet), Token-passing Bus (IEEE 802.4), Token-ring (IEEE 802.5), HDLC, and DDCMP. It is strongly recommended that a console implementation support both BOOTP-UDP/IP and MOP protocols over all supported network devices and data links.

3.8 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Removed references to ELN
2. ULTRIX –> DEC OSF/1
3. Widget –> device
4. Added eco #30 text part
5. Material rearranged according to SRM Rev 5 requirements
6. Added ECO #17, #23
7. Converted to SDML.
8. Replace previous Console Chapter with Console ECO #15
9. Includes 3 chapters and two appendices, renumber I/O Chapter
10. Material substantially changed or rearranged

\

Appendixes

The following appendixes are included in the *Alpha System Reference Manual*:

- Appendix A, Software Considerations
- Appendix B, IEEE Floating-Point Conformance
- Appendix C, Instruction Encodings
- Appendix D, Registered System and Processor Identifiers
- Appendix E, Registered Console Implementation Functions

Contents

Appendixes

Appendix A Software Considerations

Appendix B IEEE Floating-Point Conformance

Appendix C Instruction Encodings

Appendix D Registered System and Processor Identifiers

Appendix E Registered Console Implementation Functions

Figures

Tables

Appendix A
Software Considerations

A.1 Hardware-Software Compact

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

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

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

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

As an architecture becomes commercially successful, high-end implementations inevitably move to double-width data paths that can transfer an *aligned* (at an even natural word address) pair of natural words in one cycle. For Alpha, this means eventual 128-bit wide data paths. It is hard 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 hard to retrofit to old code, the following sections sometimes encourage "over-aligning" to octaword boundaries in anticipation of high-speed Alpha implementations.

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

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

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

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

A.2 Instruction-Stream Considerations

The following sections describe considerations for the instruction stream.

A.2.1 Instruction Alignment

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

Most Alpha implementations will fetch aligned quadwords of instruction stream (two instructions), and many will waste an instruction-issue cycle on a branch to an odd longword. High-end implementations may eventually fetch aligned octawords, and waste up to 3 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 2 or even 3 cache misses.

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

A.2.2 Multiple Instruction Issue — Factor of 3

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

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.

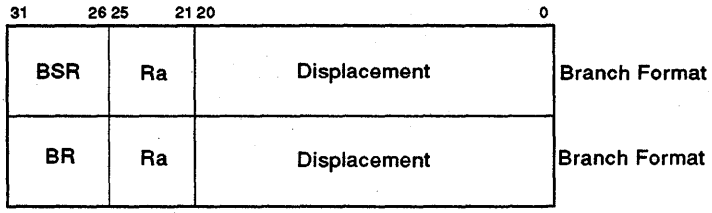
A.2.3 Branch Prediction and Minimizing Branch-Taken — Factor of 3

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

Compilers should follow these rules to minimize unexpected branches:

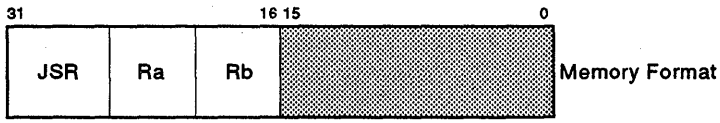
1. Implementations will predict all forward conditional branches as not-taken, and all backward conditional branches as taken. Based on execution profiles, compilers should physically rearrange code so that it has matching behavior.
2. Make basic blocks as big as possible. A good goal is 20 instructions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

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



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

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



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

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

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

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

Implementors should give first priority to executing straight-line code with no branch-takens as quickly as possible, second priority to predicting conditional branches based on the sign of the displacement field (backward taken, forward not-taken), and third priority to predicting subroutine return addresses by running a small prediction stack. (VAX traces show a stack of 2 to 4 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 percent 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 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 ......................IiiiIiiIiiIiiiiiiiiIiii..............
00026900 ..................................................IiiiiIiiiiiiiii
00026940 IiIiiIiIiiIiIiIiIiiiIiIiiIiIiiiiiiiIiiIiii.....................
00026980 ..............................................IiiiIiiIiiIiiIiiIiIiiIii
000269C0 I.............IiiiiIiiIiiiiIiIiiiiIiiiIIiIiIiiIiIiiiIiIiii.....
00026A00 ............................IiIiiiiiiiiiiiiiIiiIiiiIiii.........
00026A40 ................................Iiiiiiiiii Iiiiiiiii IiIiiiIiiIii
00026A80 IiIiiiiIiIiIiiiIiIiIiIiiiiiiiiIiiIiiiIiii.................IiiIii
00026AC0 IiiIiii.....................................................

80004440 ..............................................IiiiIiIiii.........
80004680 ....Iiiiii Iiii.....................................................
80004900 ................IiiIiiIiiIiiiiIiIiiIiiIiiIiiiiIiIiiiiIiIiiiIiiiiI
80004940 IiiiiIiiiIiiIiIiii.............IiiiiIiii.............................
80004A00 ..................................................IiiiiiIiiIiiiii
80004A40 IiIiiIiiiiIiiiIiiiIiiiIiii.............IiiiiiIIiiiiiIiiiiIiiIiiiI
80004A80 IiiiiIiiiIiiIiiIiii....IiiIiiIiii.....................................
80004F40 ............................IiiiiiIiiiiiiIiiiIiiiiiIiii.......
80004F80 ......................IiiiIiiiiiiiIiiIiIiiiIiiiiiiiiiiiiiiIiiiI
80004FC0 IIiiiiiIiiiIiIiiiIiii.....IiiiiIiIiii.............................
80008A40 ..................................................IiiiIiii
80008A80 IIiiIiiiIiiIiIiiiIiIiIiiiIiIiiIiiiiiIiiIiiIiiIiiiiiiiIiIiiiIiii.
```

A.2.5 Instruction Scheduling — Factor of 3

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

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

Most Alpha instructions have a non-zero result latency. Compilers should schedule code so that a result is not used too soon, at least in frequently executed code (inner

loops, as identified by execution profiles). In general, this will require loop unrolling and short procedure inlining.

"Too soon" is currently ill-defined, since no implementations have been designed yet. For starters, 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 will give at least reasonable performance on currently anticipated implementations. \More precise tables will be supplied in later versions of this Appendix, as the information becomes available.\

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 this. 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 writeback caches that read a partially filled cache block from memory, but optimize away the read for completely filled blocks.

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

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

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

Compiled code for parameters should assume that the parameters are aligned. Unaligned actuals will therefore cause runtime 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 4-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 4 processors work on every fourth array element severely impairs performance on any implementation with a cache block of 4 elements or larger. The processors all contend for copies of the *same* cache blocks and use only 1/4 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 4 processors and row-major array storage; N/4 columns for column-major storage). With the possible exception of 3 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

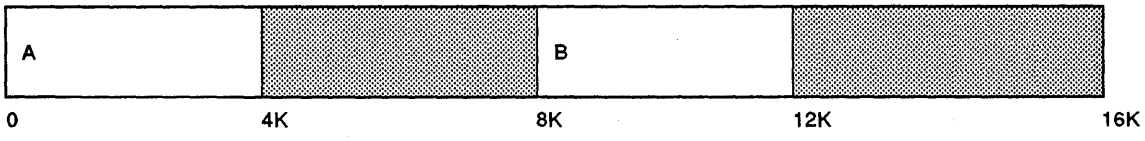
Occasionally, programs that run with a direct-mapped cache or TB will thrash, taking excessive cache or TB misses. With some work, thrashing can be minimized at compile time.

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.

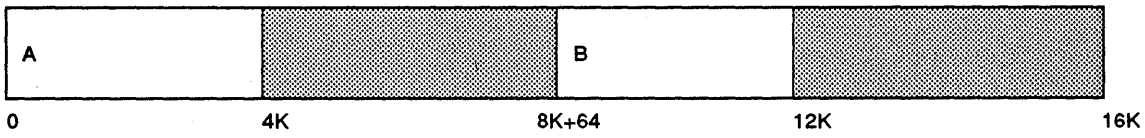
Example:

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

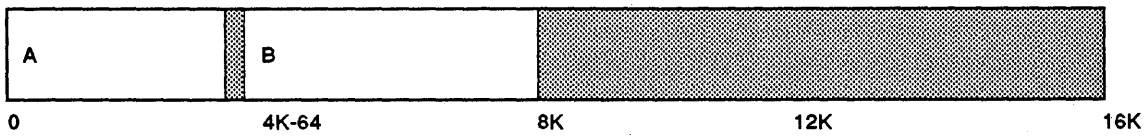
BAD allocation (A and B thrash in 8 KB direct-mapped cache):



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 Loads or Stores 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 Loads (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 Stores (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 4 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 4 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 5 (compared to a best case of 1 and a worst case of 9).

A.4.3 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.4.3.1 NOP

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, no destination issue delay, and no functional unit issue delay. Implementations are free to optimize these into no action and zero execution cycles.

A.4.3.2 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.4.3.3 Load Literal

The standard load integer literal (ZEXT 8-bit) form is:

```
MOV #lit8,Ry    ==    BIS R31, lit8, Ry
```

The Alpha literal construct in Operate instructions creates a canonical longword constant for values 0..255.

A longword constant stored in an Alpha 64-bit register is in canonical form when bits <63:32>=bit <31>.

A canonical 32-bit literal can usually be generated with one or two instructions, but sometimes three instructions are needed. Use the following procedure to determine the offset fields of the instructions:

```
val  = <sign-extended, 32-bit value>

low  = val<15:0>
tmp1 = val - SEXT(low)   ! Account for LDA instruction

high = tmp1<31:16>
tmp2 = tmp1 - SHIFT_LEFT( SEXT(high,16) )

if tmp2 NE 0 then
     ! original val was in range 7FFF8000[16]..7FFFFFFF[16]
           extra = 4000[16]
           tmp1 = tmp1 - 40000000[16]
           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.4.3.4 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.4.3.5 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.4.3.6 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.4.3.7 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.4.4 Trap Barrier

The TRAPB instruction guarantees that 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.4.5 Pseudo-Operations (Stylized Code Forms)

This section summarizes the pseudo-operations for the Alpha architecture that may be used by various software components in an Alpha system. Most of these forms are discussed in preceding sections.

In the context of this section, pseudo-operations all represent a single underlying machine instruction. Each pseudo-operation represents a particular instruction with either replicated fields (such as FMOV), or hard-coded zero fields. Since the pattern is distinct, these pseudo-operations can be decoded by instruction decode mechanisms.

In Table A–1, the pseudo-operation codes can be viewed as macros with parameters. The formal form is listed in the left column, and the expansion in the code stream listed in the right column.

Some instruction mnemonics have synonyms. These are different from pseudo-operations in that each synonym represents the same underlying instruction with no special encoding of operand fields. As a result, synonyms cannot be 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)

Table A–1 (Cont.): Decodable Pseudo-Operations (Stylized Code Forms)

Pseudo-Operation in Listing	Actual Instruction Encoding
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: 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

Table A–1 (Cont.): Decodable Pseudo-Operations (Stylized Code Forms)

Pseudo-Operation in Listing	Actual Instruction Encoding
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

A.5 Timing Considerations: Atomic Sequences

A sufficiently long instruction sequence between LDx_L and STx_C will never complete, because periodic timer interrupts will always occur before the sequence completes. The following rules describe sequences that will eventually complete in all Alpha implementations:

1. At most 40 operate or conditional-branch (not taken) instructions executed in the sequence between LDx_L and STx_C.
2. At most two I-stream TB-miss faults. Sequential instruction execution guarantees this.
3. No other exceptions triggered during the last execution of the sequence.

IMPLEMENTATION NOTE

On all expected implementations, this allows for about 50 μsec of execution time, even with 100 percent cache misses. This should satisfy any requirement for a 1 msec timer interrupt rate.

A.6 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Changed cache block sizes
2. Changed DRAINT to TRAPB
3. Converted to SDML
4. Changed MOVQ to MOV for standard load 16 bit literal
5. Changed NEGS and NEGT instruction qualifiers to match SUBS and SUBT qualifiers
6. Modified text describing creation of canonical longword constants

Revision 4.0, August 21, 1991

1. Added Pseudo-op table
2. Typos
3. Change text describing JSR to indicate that PC+displacement*4 calculation will produce the low 16 bits of most likely LW target address
4. Change name of NEGz form that operates on F, D, G, S, or T floating types to FNEGz
5. Correct Load Literal code form description of sign-extended 32 bit load.
6. Added floating point data format types to 'Negate' section

Revision 3.0, March 2, 1990

1. Add section on prefetch instructions
2. Minor cleanups to match opcodes in rest of document

Revision 2.0, October 4, 1989

1. Renumber R0 as R31, F0 as F31
2. Show new byte inserts
3. Change Freeze-Thaw to LDQ/L-STQ/C

Revision 1.0, May 23, 1989

1. Reorder and add hardware implementation priorities
2. Add aligned byte/word section
3. Add stylized code form section
4. Add timing considerations section

Revision 0.0, March 15, 1989

1. Initial version

Appendix B
IEEE Floating-Point Conformance

A subset of IEEE Standard for Binary Floating-Point Arithmetic (754-1985) is provided in the Alpha floating-point instructions. This appendix describes how to construct a complete IEEE implementation.

The order of presentation parallels the order of the IEEE specification.

B.1 Alpha Choices for IEEE Options

Alpha supports IEEE single and double formats. Optional extended double is not supported.

Alpha hardware supports normal and chopped IEEE rounding modes. IEEE plus infinity and minus infinity rounding modes can be implemented in hardware or software.

Alpha hardware does not support optional IEEE software trap enable/disable modes; see the following discussion about software support.

Alpha hardware supports add, subtract, multiply, divide, convert between floating formats, convert between floating and integer formats, and compare. Software routines support square root, remainder, round to integer in floating-point format, and convert binary to/from decimal.

In the Alpha architecture, copying without change of format is not considered an operation. (LDx, CPYSx, and STx do not check for non-finite numbers; an operation would.) Compilers may generate ADDx F31,Fx,Fy to get the opposite effect.

Optional operations for differing formats are not provided.

The Alpha choice is that the accuracy provided will meet or exceed IEEE standard requirements. It is implementation-dependent whether the software binary/decimal conversions beyond 9 or 17 digits treat any excess digits as zeros.

Overflow and underflow, NaNs, and infinities encountered during software binary to decimal conversion return strings that specify the conditions. Such strings can be truncated to their shortest unambiguous length.

Alpha hardware supports comparisons of same-format numbers. Software supports comparisons of different-format numbers.

In the Alpha architecture, results are true-false in response to a predicate.

Alpha hardware supports the required six predicates and the optional unordered predicate. The other 19 optional predicates can be constructed from sequences of two comparisons and two branches.

Alpha hardware supports infinity arithmetic only by trapping 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.

Alpha hardware supports NaNs only by trapping when a NaN operand is encountered and when a NaN is to be created. A software trap handler (interposed between the hardware and the IEEE user) provides correct Signaling and Quiet NaN behavior.

In the Alpha architecture, Quiet NaNs do not afford retrospective diagnostic information.

In the Alpha architecture, copying a Signaling NaN without a change of format does not signal an invalid exception (LDx, CPYSx, and STx do not check for non-finite numbers). Compilers may generate ADDx F31,Fx,Fy to get the opposite effect.

Alpha hardware fully supports negative zero operands, and follows the IEEE rules for creating negative zero results.

Alpha hardware does not supply IEEE exception trap behavior; the hardware traps are a superset of the IEEE-required conditions. A software trap handler (interposed between the hardware and the IEEE user) provides correct IEEE exception behavior.

In the Alpha architecture, tininess is detected by hardware after rounding, and loss of accuracy is detected by software as an inexact result.

In the Alpha architecture, user trap handlers will be supported by compilers and a software trap handler (interposed between the hardware and the IEEE user), as described in the next section.

B.2 Alpha Hardware Support of Software Exception Handlers

In Alpha instructions, hardware trap behavior is determined only at compile time; short of recompiling, there are no dynamic facilities for changing hardware trap behavior.

There is an essential disparity between the Alpha design goal of fast execution and the IEEE design goal of exact trap behavior. The Alpha hardware architecture provides means for users to choose various degrees of IEEE compliance, at appropriate performance cost.

Instructions compiled without the /Software modifier cannot produce IEEE-compliant trap behavior, nor can they provide IEEE-compliant non-finite arithmetic. Trapping and stopping on non-finite operands or results (rather than the IEEE default of continuing with NaNs propagated) is an Alpha value-added behavior that some users prefer.

Instructions compiled without the /Underflow hardware trap enable modifier cannot produce IEEE-compliant underflow trap behavior, nor can they provide IEEE-compliant denormal results. They are fast and provide true zero (not minus zero) results whenever underflow occurs. This is an Alpha value-added behavior that some users prefer.

Instructions compiled without the /Inexact hardware trap enable modifier cannot produce IEEE-compliant inexact trap behavior. Trapping on Inexact will be painfully slow; few users appear to prefer this, but they can get it if they really want it.

IEEE floating-point instructions compiled with the /Software modifier produce hardware traps and unpredictable values; a software trap handler may then produce all IEEE-required behavior.

IEEE floating-point instructions compiled with the /Underflow enable modifier produce hardware traps and true zero values for underflow; a software trap handler may then produce all IEEE-required behavior.

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.

Also, users must insert a TRAPB in every basic block with a floating operation that can potentially trap, so that a software handler has an opportunity to scale the true result by 2**192 or 2**1536, as appropriate for enabled user trap handlers; and to supply the default +/– infinity, +/–MAX, +/–MIN, denormal, or zero as appropriate for disabled user trap handlers.

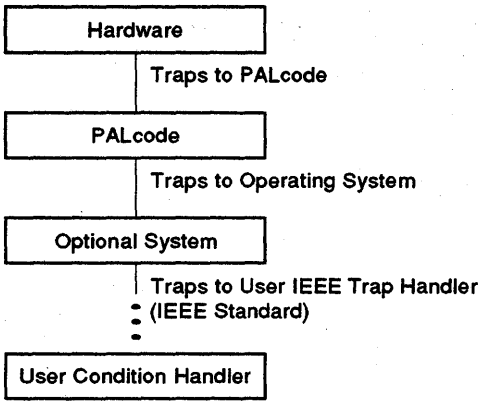
B.3 Mapping to IEEE Standard

There are five IEEE exceptions, each of which can be "IEEE software trap-enabled" or disabled (the default condition). Implementing the IEEE software trap-enabled mode is optional in the IEEE standard.

Our assumption, therefore, is that the only access to IEEE-specified software trap-enabled results will be generated in assembly language code. The following design allows this, but *only* if such assembly language code has TRAPB instructions after each floating-point instruction, and generates the IEEE-specified scaled result in a trap handler by emulating the instruction that was trapped by hardware overflow /underflow detection, using the original operands.

There is a set of detailed IEEE-specified result values, both for operations that are specified to raise IEEE traps and those that do not. This behavior is created on Alpha by four layers of hardware, PALcode, the operating-system trap handler, and the user IEEE trap handler, as shown in Figure B–1.

Figure B–1: 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–1); any trap-and-fixup behavior in the first three layers is outside the scope of the IEEE standard.

The IEEE number system is divided into finite and non-finite numbers:

- The finites are normal numbers:

  –MAX..–MIN, –0, 0, +MIN..+MAX

- The non-finites are:

  Denormals, +/– Infinity, Signaling NaN, Quiet NaN

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

Table B–1 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–1: IEEE Floating-Point Trap Handling

Alpha 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 2**Alpha
Exponent underflow and disabled	Supply +0	–	–	–[1]
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by 2**Alpha
Inexact and disabled in the instruction	–	–	–	–
Inexact and enabled in the instruction	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–1 (Cont.): IEEE Floating-Point Trap Handling

Alpha 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 2**Alpha
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and Trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by 2**Alpha
Inexact and disabled	–	–	–	–
Inexact and enabled	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]
0/0 or Inf/Inf	Trap	Trap	Supply QNaN	[Invalid Op]

Table B–1 (Cont.): IEEE Floating-Point Trap Handling

Alpha Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
DIVx INPUT Exceptions				
A/0	Trap	Trap	Supply +/–Inf	[Div. Zero]
DIVx OUTPUT Exceptions				
Exponent overflow	Trap	Trap	Supply +/–Inf +/–MAX	[Overflow] Scale by 2**Alpha
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by 2**Alpha
Inexact and disabled	–	–	–	–
Inexact and enabled	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 (=)	–
QNaN operand	Trap	Trap	Supply False	[Invalid Op]
SNaN operand	Trap	Trap	Supply False	[Invalid Op]

Table B–1 (Cont.): IEEE Floating-Point Trap Handling

Alpha Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
CVTFi INPUT Exceptions				
Denormal operand	Trap	Trap	Supply Cvt	–
+/-Inf operand	Trap	Trap	Supply Cvt	[Invalid Op]
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
CVTFi OUTPUT Exceptions				
Inexact and disabled	–	–	–	–
Inexact and enabled	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	Trap	Trap	–	[Inexact]
CVTff INPUT Exceptions				
Denormal operand	Trap	Trap	Supply Cvt	–
+/-Inf operand	Trap	Trap	Supply Cvt	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]

[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–1 (Cont.): IEEE Floating-Point Trap Handling

Alpha Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
CVTff OUTPUT Exceptions				
Exponent overflow	Trap	Trap	Supply +/–Inf +/–MAX	[Overflow] Scale by 2**Alpha
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by 2**Alpha
Inexact and disabled	–	–	–	–
Inexact and enabled	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–2 shows the IEEE standard charts.

Table B–2: 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, Inf, NaN)	Quiet NaN	
(8) 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	0/denorm/+ –MIN	Res*2**192 or 1536
Inexact	Rounded/ovfl	Res

IEEE software trap handler requirements are as follows:

Result is unpredictable unless supplied by trap handler.
Determine which exceptions occurred.
Determine the kind of operation.
Determine the destination format.
Overflow/underflow/inexact: the correctly rounded result, including parts that do not fit in the format.
Invalid and divzero: the operand values.

B.4 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Reconciled TBDs
2. Changed DRAINT to TRAPB
3. Converted to SDML

Revision 4.0, August 21, 1990

1. Remove input exceptions for –0. This should have been removed in revision 3.0
2. Typos
3. Change 'IEEE user' to 'user IEEE' in section Mapping to IEEE Standard
4. Specified T floating point data type for CMP instructions and eliminated '+/–Inf operand' input exception from these instructions

Revision 3.0, March 2, 1990

1. Revise and simplify IEEE trap behavior

Revision 2.0, October 4, 1989

1. Initial version

\

Appendix C
Instruction Encodings

The encodings for the Alpha instruction set are given in the following sections. There is one section for each instruction format, followed by a summary of all the instruction opcodes in a single table.

NOTE

\ To receive a VAX Structure Definition Language (SDL) file defining the opcodes and function codes send mail to AD::Alpha_$OPCODES.\

C.1 Memory Format Instructions

Table C–1 lists the hexadecimal values of the 6-bit opcode field for the Memory format instructions.

Table C–1: Memory Format Instruction Opcodes

Mnemonic		Mnemonic		Mnemonic	
LDA	08	LDAH	09	LDF	20
LDG	21	LDL	28	LDL_L	2A
LDQ	29	LDQ_L	2B	LDQ_U	0B
LDS	22	LDT	23	STF	24
STG	25	STL	2C	STL_C	2E
STQ	2D	STQ_C	2F	STQ_U	0F
STS	26	STT	27		

Table C–2 lists the hexadecimal values of the 6-bit opcode field and the 16-bit displacement field for the Memory format instructions that use the displacement field as a function code. The notation used is oo.ffff, where *oo* is the 6-bit opcode and the *ffff* is the 16-bit displacement field.

Table C–2: Memory Format Instructions with a Function Code

Mnemonic		Mnemonic		Mnemonic	
FETCH	18.8000	FETCH_M	18.A000	MB	18.4000
RC	18.E000	RPCC	18.C000	RS	18.F000
TRAPB	18.0000				

PROGRAMMING NOTE

The code points 18.4400, 18.4800, and 18.4C00 must operate as Memory Barrier instructions (MB 18.4000). Software will currently only use the 18.4000 code point for MB. This allows a weaker memory barrier to be added.

Table C–3 lists the hexadecimal values of the high-order two bits of the displacement field for the Memory format branch instructions. The notation used is oo.h, where *oo* is the 6-bit opcode and the *h* is the high-order two bits of the displacement field.

Table C–3: Memory Format Branch Instruction Opcodes

Mnemonic		Mnemonic		Mnemonic	
JMP	1A.0	JSR	1A.1	JSR_COROUTINE	1A.3
RET	1A.2				

C.2 Branch Format Instructions

Table C–4 lists the hexadecimal values of the 6-bit opcode field for the Branch format instructions.

Table C–4: Branch Format Instruction Opcodes

Mnemonic		Mnemonic		Mnemonic	
BR	30	FBEQ	31	FBLT	32
FBLE	33	BSR	34	FBNE	35
FBGE	36	FBGT	37	BLBC	38
BEQ	39	BLT	3A	BLE	3B
BLBS	3C	BNE	3D	BGE	3E
BGT	3F				

C.3 Operate Format Instructions

Table C–5 lists the hexadecimal values of the 6-bit opcode field and the 7-bit function code field for the Operate format instructions The notation used is oo.ff, where *oo* is the 6-bit opcode and the *ff* is the 7-bit function code field

Table C–5: Operate Format Instruction Opcodes and Function Codes

Mnemonic		Mnemonic		Mnemonic	
ADDL	10.00	ADDL/V	10.40	ADDQ	10.20
ADDQ/V	10.60	CMPBGE	10.0F	CMPEQ	10.2D
CMPLE	10.6D	CMPLT	10.4D	CMPULE	10.3D
CMPULT	10.1D	SUBL	10.09	SUBL/V	10.49

Table C–5 (Cont.): Operate Format Instruction Opcodes and Function Codes

Mnemonic		Mnemonic		Mnemonic	
SUBQ	10.29	SUBQ/V	10.69		
S4ADDL	10.02	S4ADDQ	10.22	S4SUBL	10.0B
S4SUBQ	10.2B	S8ADDL	10.12	S8ADDQ	10.32
S8SUBL	10.1B	S8SUBQ	10.3B		
AND	11.00	BIC	11.08	BIS	11.20
CMOVEQ	11.24	CMOVLBC	11.16	CMOVLBS	11.14
CMOVGE	11.46	CMOVGT	11.66	CMOVLE	11.64
CMOVLT	11.44	CMOVNE	11.26	EQV	11.48
ORNOT	11.28	XOR	11.40		
EXTBL	12.06	EXTLH	12.6A	EXTLL	12.26
EXTQH	12.7A	EXTQL	12.36	EXTWH	12.5A
EXTWL	12.16	INSBL	12.0B	INSLH	12.67
INSLL	12.2B	INSQH	12.77	INSQL	12.3B
INSWH	12.57	INSWL	12.1B	MSKBL	12.02
MSKLH	12.62	MSKLL	12.22	MSKQH	12.72
MSKQL	12.32	MSKWH	12.52	MSKWL	12.12
SLL	12.39	SRA	12.3C	SRL	12.34
ZAP	12.30	ZAPNOT	12.31		
MULL	13.00	MULL/V	13.40	MULQ	13.20
MULQ/V	13.60	UMULH	13.30		

C.4 Floating-Point Operate Format

Table C–6 lists the hexadecimal values of the 11-bit function code field for the Floating-point Operate format instructions that are data type independent. The 6-bit opcode for these instructions is 17_{16}.

Table C–6: Function Codes for Floating Data Type Independent Operations

Mnemonic		Mnemonic		Mnemonic	
CPYS	020	CPYSE	022	CPYSN	021
CVTLQ	010	CVTQL	030	CVTQL/SV	530
CVTQL/V	130				
FCMOVEQ	02A	FCMOVGE	02D	FCMOVGT	02F
FCMOVLE	02E	FCMOVLT	02C	FCMOVNE	02B
MF_FPCR	025	MT_FPCR	024		

C.4.1 IEEE Floating-Point Instructions

Table C–7 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–7: 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				
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	/C	/V	/VC	/SV	/SVC	/SVI	/SVIC
CVTTQ	0AF	02F	1AF	12F	5AF	52F	7AF	72F

Table C–7 (Cont.): IEEE Floating-Point Instruction Function Codes

	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.

C.4.2 VAX Floating-Point Instructions

Table C–8 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–8: 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
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.5 Opcode Summary

Table C–9 lists all Alpha opcodes from 00 (CALL_PALL) through 3F (BGT). In the table, the column headings appearing 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 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_{16} column contains the symbol INTS*, representing the all integer subtract 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_{16} 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–9 are explained in Table C–10.

Table C–9: 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–10: Key to Opcode Summary (Table C–9)

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 subtract 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.6 OpenVMS PALcode Format Instructions

Sections C.6.1 and C.6.2 list the OpenVMS Alpha unprivileged and privileged PALcode function codes.

C.6.1 Unprivileged OpenVMS PALcode Function Codes

Table C–11 lists the hexadecimal values of the 26-bit function code field for the unprivileged OpenVMS PALcode format instructions. The 6-bit opcode for the PALcode instructions is zero.

Table C–11: Unprivileged OpenVMS PALcode Function codes

Mnemonic		Mnemonic		Mnemonic	
AMOVRM	00A1	AMOVRR	00A0	BPT	0080
BUGCHK	0081	CHME	0082	CHMK	0083
CHMS	0084	CHMU	0085	GENTRAP	00AA
IMB	0086	INSQHIL	0087	INSQHILR	00A2
INSQHIQ	0089	INSQHIQR	00A4	INSQTIL	0088
INSQTILR	00A3	INSQTIQ	008A	INSQTIQR	00A5
INSQUEL	008B	INSQUEL/D	008D	INSQUEQ	008C
INSQUEQ/D	008E	PROBER	008F	PROBEW	0090
RD_PS	0091	READ_UNQ	009E	REI	0092
REMQHIL	0093	REMQHILR	00A6	REMQHIQ	0095
REMQHIQR	00A8	REMQTIL	0094	REMQTILR	00A7
REMQTIQ	0096	REMQTIQR	00A9	REMQUEL	0097
REMQUEL/D	0099	REMQUEQ	0098	REMQUEQ/D	009A
RSCC	009D	SWASTEN	009B	WRITE_UNQ	009F
WR_PS_SW	009C				

C.6.2 Privileged OpenVMS PALcode Function Codes

Table C–12 lists the hexadecimal values of the 26-bit function code field for the privileged OpenVMS PALcode format instructions. The 6-bit opcode for the PALcode instructions is zero.

Table C–12: Privileged OpenVMS PALcode Function Codes

Mnemonic		Mnemonic		Mnemonic	
CFLUSH	0001	DRAINA	0002	HALT	0000
LDQP	0003				
MFPR_ASN	0006	MFPR_ASTEN	0026	MFPR_ASTSR	0027
MFPR_ESP	001E	MFPR_FEN	000B	MFPR_IPL	000E
MFPR_MCES	0010	MFPR_PCBB	0012	MFPR_PRBR	0013
MFPR_PTBR	0015	MFPR_SCBB	0016	MFPR_SISR	0019
MFPR_SSP	0020	MFPR_TBCHK	001A	MFPR_USP	0022
MFPR_VPTB	0029	MFPR_WHAMI	003F		

Table C–12 (Cont.): Privileged OpenVMS PALcode Function Codes

Mnemonic		Mnemonic		Mnemonic	
MTPR_ASTEN	0007	MTPR_ASTSR	0008	MTPR_DATFX	002E
MTPR_ESP	001F	MTPR_FEN	000C	MTPR_IPIR	000D
MTPR_IPL	000F	MTPR_MCES	0011	MTPR_PERFMON	002B
MTPR_PRBR	0014	MTPR_SCBB	0017	MTPR_SIRR	0018
MTPR_SSP	0021	MTPR_TBIA	001B	MTPR_TBIAP	001C
MTPR_TBIS	001D	MTPR_TBISD	0024	MTPR_TBISI	0025
MTPR_USP	0023	MTPR_VPTB	002A		
STQP	0004	SWPCTX	0005	unused	0009
unused	000A				

C.7 Unprivileged OSF/1 PALcode Function Codes

Table C–13 lists lists the hexadecimal values of the 26-bit function code field for the unprivileged OSF/1 PALcode instructions. The 6-bit opcode for the PALcode instructions is zero.

Table C–13: Unprivileged OSF/1 PALcode Function Codes

Mnemonic		Mnemonic		Mnemonic	
bpt	0080	bugchk	0081	callsys	0083
gentrap	00AA	imb	0086	rdunique	009E
wrunique	009F				

C.8 Privileged OSF/1 PALcode function codes

Table C–14 lists lists the hexadecimal values of the 26-bit function code field for the unprivileged OSF/1 PALcode instructions. The 6-bit opcode for the PALcode instructions is zero.

Table C–14: Privileged OSF/1 PALcode Function Codes

Mnemonic		Mnemonic		Mnemonic	
halt	0000	rdps	0036	rdusp	003A
rdval	0032	retsys	003D	rti	003F
swpctx	0030	swpipl	0035	tbi	0033
whami	003C	wrent	0034	wrfen	002B
wrkgp	0037	wrusp	0038	wrval	0031
wrvptptr	002D				

C.9 Required PALcode Function Codes

The opcodes listed in Table C–15 are required for all Alpha implementations. The notation used is oo.ffff, where *oo* is the hexadecimal 6-bit opcode and *ffff* is the hexadecimal 26-bit function code.

Table C–15: Required PALcode Function Codes

Mnemonic	Type	Function Code
DRAINA	Privileged	00.0002
HALT	Privileged	00.0000
IMB	Unprivileged	00.0086

C.10 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.11 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				

\PROGRAMMING NOTE (SRM ONLY)

Opcodes 02, 06, 0A, and 0E are nominally reserved for future extensions to octaword load/store for both integer and floating-point formats.

For IEEE Floating-point opcode 16_{16}, if the function code field bits<5:4> are 01_2 or the function code bits<3:0> are 1101_2, then an illegal instruction trap

is taken. This will allow for future additions of the extended IEEE format. \

C.12 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Added note on IEEE floating-point code 16, special function code fields
2. Added DRAINA to list of required PALcode instructions
3. Added ECO #17, #23
4. Converted to SDML
5. Removed /S and /SC opcodes from CVTQF and CVTQG instructions encodings
6. Corrected text by removing extra 'instructions' from Fig. C-3 text
7. Added CMPBGE to Operate format instruction encoding
8. Add opcode for READ_UNQ and WRITE_UNQ

Revision 4.0, March 29, 1991

1. Changed /P to /D
2. Added RSCC opcode
3. Added Scaled Add/Subtract opcodes
4. Removed references to D_float
5. Updated various opcodes per EV-4 request
6. Typos

Revision 3.0, Mar 2, 1990

1. Version 3.0 update

Revision 2.0, October 4, 1989

1. First Pass

\

Appendix D
Registered System and Processor Identifiers

This appendix contains a registry of Alpha system platform types, system platform variations, processor types, processor variations, and processor packaging types. See *Platform Section, Chapter 3* for a description of these fields.

\ Send mail to EAGLE1::ALPHA_SRM to register a new Alpha system, platform, or processor. Note that the Alpha system types are not equivalent to the VAX SYSTYPE values.\

Table D–1: System and Processor Identification Assignments

System Type	Processor Type	Product Name
1 ADU	1 = EV-3 2 = EV-4	
2 Cobra	1 = EV-3 2 = EV-4	
3 Ruby	1 = EV-3 2 = EV-4	
4 Flamingo	1 = EV-3 2 = EV-4	
5 Mannequin	3 = Simulation	
6 Jensen	2 = EV_4	

Table D–2: System Variation Assignments

Bit	Description
0	MPCAP - If set, indicates this system platform is capable of being configured as a multiprocessor; all support for multiprocessing is present, even if only one processor is present. If clear, this system supports a uniprocessor only. Initialized by the console at all cold bootstraps.
4:1	CONSOLE - Indicates the type of console. Defined values include: **<4:1> Interpretation** 0000 Reserved 0001 Detached service processor 0010 Embedded console other Reserved for future use Initialized by the console at all cold bootstraps.
7:5	POWERFAIL - Indicates the type of powerfail (if any) implemented by this platform. Defined values include: **<7:5> Interpretation** 000 Reserved 001 United 010 Separate 011 Full battery backup of system platform hardware Initialized by the console at all cold bootstraps.
8	POWERFAIL RESTART - If set, indicates that the console should restart all available processors on a powerfail recovery. If clear, only the primary processor will be restarted. Cleared by the console at system bootstraps; may be set by system software.
9	GRAPHICS - If set, indicates that the platform contains an imbedded graphics processor. Initialized by the console at all cold bootstraps.
63:10	RESERVED - MBZ

Table D–3: Processor Variation Assignments

Bit	Description
0	VAX-FP - If set, indicates this processor supports VAX Floating-point operations and data types. If clear, this processor has no such support. Initialized by the console at all cold bootstraps.
1	IEEE-FP - If set, indicates this processor supports IEEE Floating-point operations and data types. If clear, this processor has no such support. Initialized by the console at all cold bootstraps.
2	PRIMARY ELIGIBLE (PE) - If set, indicates that this processor is eligible to become a primary processor. The processor has direct access to the console, a BB_WATCH, and all I/O widgets. Initialized by the console at all cold bootstraps. See *Platform Section, Chapter 4.*
63:3	RESERVED - MBZ

D.1 I/O Architecture Section

This section includes that information removed from the I/O chapter previously located in the Platforms section.

D.1.1 Special Commands

The special "WHO_ARE_YOU" command (W=0, B=1, CMD=0) is common to all bridge implementations. WHO_ARE_YOU is used to determine the type of remote bridge side. In response to a mailbox operation with a WHO_ARE_YOU command and RBADR of 0, the remote bridge side returns a unique remote bus side identifier. All other commands are specific to the type of remote bus and independent of the bridge implementation.

Table D–4: WHO_ARE_YOU returns

Bus	Bridge	System Type(s)	WHO_ARE_YOU returns
XMI	LAMB	Laser	XMI XDEV register
			<31:16> Device revison
			<15:0> $102A_{16}$
Futurebus+		Cobra	Not implemented
	FLAG	Laser	TBD

D.1.1.1 XMI Specific Information

The XMI CMD field definition is given in Table 11-4. Bits <39:0> of the RBADR field are passed unchanged onto the XMI by the remote side. The MASK field is inverted to form the XMI byte enables.

To access XMI device CSRs, RBADR field bit <31> must be clear and bits <30:29> must be set. Only longword accesses are supported; MASK bits <7:4> must be set and WDATA bits <63:32> are ignored by the bridge.

Table D–5: XMI CMD field

Bit(s)	Name	Description
<3:0>	TRANS	Transaction type: 0 undefined 1 read longword 2-6 undefined 7 write longword

D.1.1.2 Futurebus+ Specific Information

The Futurebus+ CMD field definition is given in *Platform Section, Chapter 1*. RBADR must be longword aligned for longword read or write accesses and quadword aligned for quadword write accesses. The MASK field is passed unchanged onto the Futurebus+ by the remote side.

Table D–6: Futurebus+ CMD field

Bit(s)	Name	Description
<3:0>	TC	Transaction code. 0 unmasked 1 undefined 2 partial - byte mask is valid 3-7 undefined
<4>	WR	Write transaction. 0 Read 1 Write
<6:5>	DW	Data width. Note that all widths may not be implemented by the remote side. 0 32-bits 1 64-bits 2 128-bits 3 256-bits Cobra and FLAG implement only 32-bit and 64-bit data widths.

Table D–6 (Cont.): Futurebus+ CMD field

Bit(s)	Name	Description
<7>	AW	Address width. 0 32-bits 1 64-bits
<22:16>	F_DIAG	FLAG specific diagnostic bits. See FLAG specification.
<29:23>	C_DIAG	Cobra specific diagnostic bits. See Cobra I/O specification.

D.2 \Revision History

Revision 5.0, May 12, 1992

1. Added XMI and Future+ tables from I/O chapter
2. Added Jensen identifier
3. Added graphics variation bit (9)

\

Appendix E

Registered Console Implementation Functions

This appendix contains a registry of functions as implemented by current consoles. The first two sections contain the registered environment variables and console terminal blocks. The remaining sections summarize the functions implemented by existing consoles.

\Console functions which vary with implementation are summarized in *Platform Section, Chapter 2*. All console implementations and all such implementation-specific functions must be registered with the Alpha Architecture Group by sending mail to EAGLE1::ALPHA_SRM.\

E.1 Environment Variables

Table E–1: Option Environment Variables

Environment Var ID	Symbol	Notes	Description
40–7F			TBD

E.2 Console Terminal Block Formats

E.2.1 Serial Line UART

Console terminal type '02' supports the full functionality of a VT device.

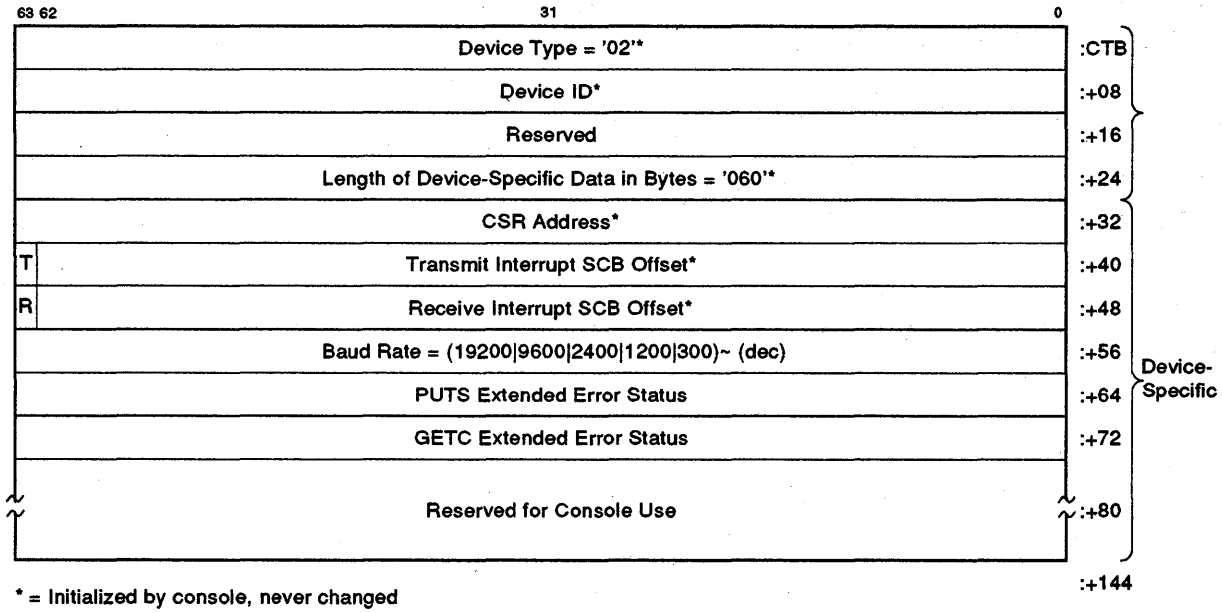
If the terminal interface is shared among multiple physical terminals, the device ID indicates which physical terminal is the console terminal. If the terminal interface is not shared, the device ID is zero.

Extended error status may result from the PUTS and GETC console callback routines. As shown above, extended status is recorded at offsets [64] and [72]; the format is:

<63:3> SBZ
<2> '1' Data Overrun
'0' otherwise
<1> '1' Framing error
'0' otherwise
<0> '1' Parity error
'0' otherwise

SET_TERM_CTL alters only the baud rate at offset [56]. Support for multiple baud rates is implementation-specific.

Figure E–1: Serial Line UART Format



* = Initialized by console, never changed
^ = Initialized by console, updated by console
~ = Initialized by console, updated by system software
T = Set to '1' if transmit interrupts enabled
R = Set to '1' if receive interrupts enabled

E.2.2 Graphic Display with LK Keyboard

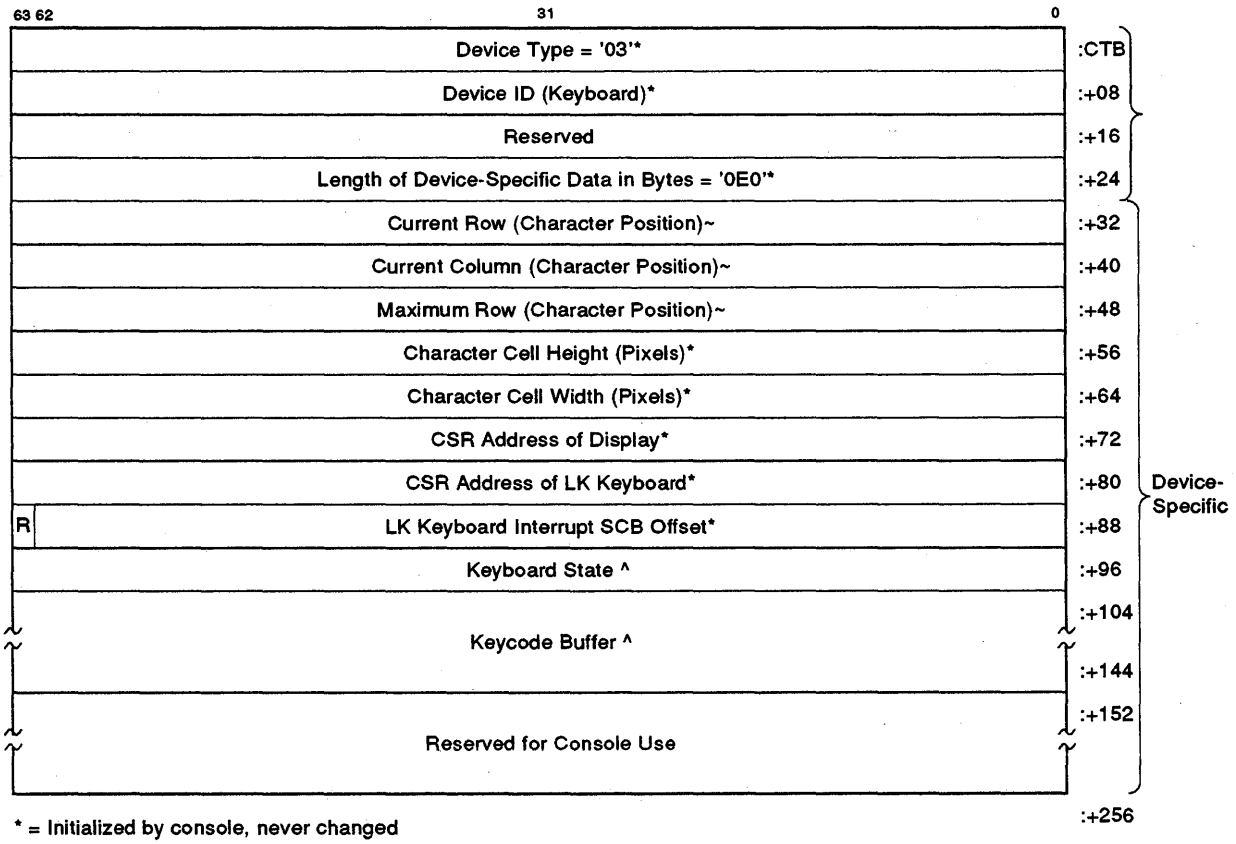
Console terminal type '03' is connected by a serial line UART and supports the LK keyboard functions as follows:

- 48 graphic keys and spacebar on the typewriter mass
- Numeric keypad
- Delete, Return, and TAB characters
- Control-character sequences
- Shift-key (uppercase) sequences
- CAPS-LOCK activation including the appropriate turning on and off of LED3, the CAPS-LOCK LED
- METRONOME code, $B4_{16}$, used for autorepeat mode
- Lighting of the LED4 (Hold Screen LED) when output flow control is enabled and active.
- Severe error keycodes

All other special keycode operations are unsupported. Unsupported functions also include the COMPOSE-key and other alternate keycode select mechanisms.

If the interface to the keyboard is shared among multiple devices (e.g. mouse), the device ID indicates the keyboard unit. If the keyboard interface is not shared, the device ID is zero.

Figure E–2: Serial Line UART with LK Keyboard Format



* = Initialized by console, never changed
^ = Initialized by console, updated by console
~ = Initialized by console, updated by system software
R = Set to '1' if receive interrupts enabled

Keyboard

State	Interpretation	Default value
<0>	Keyboard error	0, none
<1>	CTRL sequence in progress	0, none

Keyboard

State	Interpretation	Default value
<2>	Shift_key sequence in progress	0, off
<3>	CAPS LOCK in effect	0, off
<4>	Output flow control enabled	1, enabled
<5>	Output flow control status	0, inactive

The keyboard is assumed to be in LK200 mode with default settings as described in the LK400 Functional Specification, Appendix II. The default key transmission modes at power up are:

Keyboard Division	Mode
Main Array	autorepeat
Keypad	autorepeat
Del	autorepeat
Return and Tab	down only
Function keys	down only
Lock,A00,A10	down only
Shift,Ctrl,A01,A09	down up
Cursor keys	autorepeat
6 Basic Editing Keys	down only
Audio volume	keyclick and bell volumes are 2 (dec). the Ctrl (C99) and Shift (B99 and B11) keys do not not generate clicks.

A REINITIATE KEYBOARD command, FD_{16}, is sent to the keyboard when any of the following severe errors are encountered during execution of a console terminal callback routine:

1. TEST MODE ACKNOWLEDGE - $B8_{16}$
2. OUTPUT ERROR - $B5_{16}$
3. INPUT ERROR - $B6_{16}$
4. KEYBOARD LOCKED CONFIRMATION - $B7_{16}$

KEYBOARD_STATE<0> set to '1' when a POWER-UP keycode, 3D or $3E_{16}$, is received from the keyboard. While KEYBOARD_STATE<0> is set to '1', calls to GETC or PROCESS_KEYCODE for this unit fail with error status.

SET_TERM_CTL has no affect on any CTB field for this terminal device type.

E.3 Implemented Console Functions

E.3.1 Cobra and Laser Systems

The Cobra and Laser Systems share a common console firmware code base. As such, most of the implemented functions are common. Common functions are summarized below.

Table E–2: Cobra and Laser Console Functionality

Function	Description
HWRPB Version	'2'.
CTB format(s)	Serial line UART (type '02').
Optional Callbacks	PSWITCH implemented; SAVE_ENV and PROCESS_KEYCODE not implemented.
Environment Variables	No implementation-specific environment variables accessible by system software.
BOOTED_DEV format	Device path values consist of six fields as follows:

Field	Contents	
protocol	mscp scsi dssi mop	
hose	Cobra:	Local I/O: 0 FBus I/O: 1
	Laser:hose: 0-3	
slot	Cobra:	FBus node: 0-6
	Laser:	XMI node: 0-14 FBus node: 0-14
channel	Device channel number (0-n) (valid only for multiple channel widgets)	
remote_address	CI, DSSI, SCSI node number	
unit	Disk or tape unit number	

A Cobra example is "mop 1 6 1 0 0" indicating a MOP bootstrap from the first channel of an FNA at the sixth FBus node. A Laser example is "mscp 2 3 0 11 9" indicating a bootstrap from disk unit 9 on an HSC connected to CI node 11 accessed from an XCD at node 3 of an XMI connected to hose 2 of the IOP.

BOOT_DEV format	The number of list elements is TBD.

Table E–2 (Cont.): Cobra and Laser Console Functionality

Function	Description
BOOTDEF_DEV format	TBD.
BOOTED_FILE format	TBD.
BOOT_OSFLAGS format	The value consists of a list of up to four single hex digit flags. Examples are "2,7", ",7", or "c,2,4,b".
CONFIG format	See Figure TBD for Cobra. See Figure TBD for Laser.
Bootstrap media	TBD.
HALT codes	No implementation specific codes.

E.3.2 Flamingo System Console Functions

Table E–3: Flamingo Console Functionality

Function	Description
HWRPB Version	'2'.
CTB format(s)	Graphic Display with LK Keyboard (type '03').
Optional Callbacks	PROCESS_KEYCODE implemented; SAVE_ENV and PSWITCH not implemented.
Environment Variables	No implementation-specific environment variables accessible by system software.
BOOTED_DEV format	TBD.
BOOT_DEV format	TBD.
BOOTDEF_DEV format	TBD.
BOOT_OSFLAGS format	TBD.
BOOTED_FILE format	TBD.
CONFIG format	See Figure TBD.
Bootstrap media	TBD.
HALT codes	No implementation specific codes.

E.4 \REVISION HISTORY

Revision 5.0, May 12, 1992

1. Added ECO #30
2. Converted to SDML
3. Replace previous Console Chapter with Console ECO #15
4. Includes 3 chapters and two appendices, renumber I/O Chapter
5. Material substantially changed or rearranged

\

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z