Chapter 4

MIPS-Style Alpha Instruction Descriptions

4.1 Instruction Set Overview

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

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

Within each major section, closely related instructions are combined into groups and described together.

The instruction group description is composed of the following:

- The group name
- The format of each instruction in the group, which includes the name, access type, and data type of each instruction operand
- The operation of the instruction
- Exceptions specific to the instruction
- The instruction mnemonic and name of each instruction in the group

- Qualifiers specific to the instructions in the group
- A description of the instruction operation
- Optional programming examples and optional notes on the instruction

4.1.1 Subsetting Rules

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

4.1.2 Floating-Point Subsets

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

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

Software Note:

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

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

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

Note:

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

4.1.3 Software Emulation Rules

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

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

4.1.4 Opcode Qualifiers

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

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

Table 4–1: Opcode Qualifiers

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

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

4.2 Memory Integer Load/Store Instructions

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

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

Table 4–2: Memory Integer Load/Store Instructions

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

4.2.1 Load Address

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

LDA	Load Address
LDAH	Load Address High

Qualifiers:

None

Description:

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

4.2.2 Load Memory Data into Integer Register

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

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

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

Exceptions:

Access Violation

Alignment

Fault on Read

Translation Not Valid

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

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

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

Notes:

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

Implementation Notes:

- The LDWU and LDBU instructions are supported in hardware on Alpha implementations for which the AMASK instruction returns bit 0 set. LDWU and LDBU are supported with software emulation in Alpha implementations for which AMASK does not return bit 0 set. Software emulation of LDWU and LDBU is significantly slower than hardware support.
- Depending on an address space region's caching policy, implementations may read a (partial) cache block in order to do word/byte stores. This may only be done in regions that have memory-like behavior.
- Implementations are expected to provide sufficient low-order address bits and length-of-access information to devices on I/O buses. But, strictly speaking, this is outside the scope of architecture.

4.2.3 Load Unaligned Memory Data into Integer Register

Format:

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

Operation:

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

Exceptions:

Access Violation

Fault on Read

Translation Not Valid

Instruction mnemonics:

LDQ_U	Load Unaligned Quadword from Memory to Register

Qualifiers:

None

Description:

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

4.2.4 Load Memory Data into Integer Register Locked

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

CASE
  big_endian_data:  va' ← va XOR 000₂   ! LDQ_L
  big_endian_data:  va' ← va XOR 100₂   ! LDL_L
  little_endian_data: va' ← va          ! LDL_L
ENDCASE

lock_flag ← 1
locked_physical_address ← PHYSICAL_ADDRESS(va)

Ra ← SEXT((va')<31:0>)                  ! LDL_L
Ra ← (va)<63:0>                         ! LDQ_L
```

Exceptions:

Access Violation

Alignment

Fault on Read

Translation Not Valid

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

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

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

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

The sequence:

```
LDx_L
Modify
STx_C
BEQ xxx
```

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

Notes:

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

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

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

If two LDx_L instructions execute with no intervening STx_C, the second one overwrites the state of the first one. If two STx_C instructions execute with no intervening LDx_L, the second one always fails because the first clears lock_flag.

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

Implementation Notes:

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

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

4.2.5 Store Integer Register Data into Memory Conditional

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

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

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

Exceptions:

Access Violation

Fault on Write

Alignment

Translation Not Valid

Instruction mnemonics:

STL_C	Store Longword from Register to Memory Conditional
STQ_C	Store Quadword from Register to Memory Conditional

Qualifiers:

None

Description:

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

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

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

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

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

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

Notes:

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

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

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

Software Note:

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

Hardware/Software Implementation Note:

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

Implementation Notes:

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

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

4.2.6 Store Integer Register Data into Memory

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

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

 (va') ← Rav                             !STQ
 (va')<31:00> ← Rav<31:0>                !STL
 (va')<15:00> ← Rav<15:0>                !STW
 (va')<07:00> ← Rav<07:0>                !STB
```

Exceptions:

Access Violation

Alignment

Fault on Write

Translation Not Valid

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

The Ra operand is written to memory at this address. If the data is not naturally aligned, an alignment exception is generated.

Notes:

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

Implementation Notes:

- The STB and STW instructions are supported in hardware on Alpha implementations for which the AMASK instruction returns bit 0 set. STB and STW are supported with software emulation in Alpha implementations for which AMASK does not return bit 0 set. Software emulation of STB and STW is significantly slower than hardware support.
- Depending on an address space region's caching policy, implementations may read a (partial) cache block in order to do byte/word stores. This may only be done in regions that have memory-like behavior.
- Implementations are expected to provide sufficient low-order address bits and length-of-access information to devices on I/O buses. But, strictly speaking, this is outside the scope of architecture.

4.2.7 Store Unaligned Integer Register Data into Memory

Format:

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

Operation:

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

Exceptions:

Access Violation

Fault on Write

Translation Not Valid

Instruction mnemonics:

STQ_U	Store Unaligned Quadword from Register to Memory

Qualifiers:

None

Description:

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

4.3 Control Instructions

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

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

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

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

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

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

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

Table 4–3: Control Instructions Summary

Mnemonic	Operation
BEQ	Branch if Register Equal to Zero
BGE	Branch if Register Greater Than or Equal to Zero
BGT	Branch if Register Greater Than Zero
BLBC	Branch if Register Low Bit Is Clear
BLBS	Branch if Register Low Bit Is Set
BLE	Branch if Register Less Than or Equal to Zero
BLT	Branch if Register Less Than Zero

Table 4–3: Control Instructions Summary (Continued)

Mnemonic	Operation
BNE	Branch if Register Not Equal to Zero
BR	Unconditional Branch
BSR	Branch to Subroutine
JMP	Jump
JSR	Jump to Subroutine
RET	Return from Subroutine
JSR_COROUTINE	Jump to Subroutine Return

4.3.1 Conditional Branch

Format:

```
Bxx          Ra.rq,disp.al                    !Branch format
```

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

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

The test is on the signed quadword integer interpretation of the register contents; all 64 bits are tested.

4.3.2 Unconditional Branch

Format:

BxR	Ra.wq,disp.al	!Branch format

Operation:

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

Exceptions:

None

Instruction mnemonics:

BR	Unconditional Branch
BSR	Branch to Subroutine

Qualifiers:

None

Description:

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

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

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

PC-relative addressability can be established by:

```
        BR Rx,L1
L1:
```

Notes:

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

4.3.3 Jumps

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

All Jump instructions do identical operations. They only differ in hints to possible branch-prediction logic. The displacement field of the instruction is used to pass this information. The four different "opcodes" set different bit patterns in disp<15:14>, and the hint operand sets disp<13:0>.

These bits are intended to be used as shown in Table 4–4.

Table 4–4: Jump Instructions Branch Prediction

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

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

Note that the above information is used only as a hint; correct setting of these bits can improve performance but is not needed for correct operation. See Section A.2.2 for more information on branch prediction.

An unconditional long jump can be performed by:

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

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

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

4.4 Integer Arithmetic Instructions

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

Count instruction (CIX) extension implementation note:

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

The integer instructions are summarized in Table 4–5

Table 4–5: Integer Arithmetic Instructions Summary

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

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

4.4.1 Longword Add

Format:

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

Operation:

```
Rc ← SEXT( (Rav + Rbv)<31:0>)
```

Exceptions:

Integer Overflow

Instruction mnemonics:

ADDL	Add Longword

Qualifiers:

Integer Overflow Enable (/V)

Description:

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

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

4.4.2 Scaled Longword Add

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

4.4.3 Quadword Add

Format:

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

Operation:

```
Rc ← Rav + Rbv
```

Exceptions:

Integer Overflow

Instruction mnemonics:

ADDQ	Add Quadword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Ra is added to register Rb or a literal and the 64-bit sum is written to Rc.

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

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

4.4.4 Scaled Quadword Add

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

4.4.5 Integer Signed Compare

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

Notes:

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

4.4.6 Integer Unsigned Compare

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

4.4.7 Count Leading Zero

Format:

```
CTLZ        Rb.rq,Rc.wq                    ! Operate format
```

Operation:

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

Exceptions:

None

Instruction mnemonics:

CTLZ Count Leading Zero

Qualifiers:

None

Description:

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

4.4.8 Count Population

Format:

```
CTPOP       Rb.rq,Rc.wq                    ! Operate format
```

Operation:

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

Exceptions:

None

Instruction mnemonics:

CTPOP	Count Population

Qualifiers:

None

Description:

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

4.4.9 Count Trailing Zero

Format:

```
CTTZ            Rb.rq,Rc.wq                    ! Operate format
```

Operation:

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

Exceptions:

None

Instruction mnemonics:

CTTZ	Count Trailing Zero

Qualifiers:

None

Description:

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

4.4.10 Longword Multiply

Format:

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

Operation:

```
Rc ← SEXT ((Rav * Rbv)<31:0>)
```

Exceptions:

Integer Overflow

Instruction mnemonics:

MULL	Multiply Longword

Qualifiers:

Integer Overflow Enable (/V)

Description:

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

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

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

4.4.11 Quadword Multiply

Format:

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

Operation:

```
Rc ← Rav * Rbv
```

Exceptions:

Integer Overflow

Instruction mnemonics:

MULQ	Multiply Quadword

Qualifiers:

Integer Overflow Enable (/V)

Description:

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

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

4.4.12 Unsigned Quadword Multiply High

Format:

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

Operation:

```
Rc ← {Rav * U Rbv}<127:64>
```

Exceptions:

None

Instruction mnemonics:

UMULH	Unsigned Multiply Quadword High

Qualifiers:

None

Description:

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

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

Ra and Rb are unsigned: result of UMULH
Ra and Rb are signed: (result of UMULH) – Ra<63>*Rb – Rb<63>*Ra

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

4.4.13 Longword Subtract

Format:

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

Operation:

```
Rc ← SEXT ((Rav - Rbv)<31:0>)
```

Exceptions:

Integer Overflow

Instruction mnemonics:

SUBL	Subtract Longword

Qualifiers:

Integer Overflow Enable (/V)

Description:

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

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

4.4.14 Scaled Longword Subtract

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

S4SUBL	Scaled Subtract Longword by 4
S8SUBL	Scaled Subtract Longword by 8

Qualifiers:

None

Description:

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

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

4.4.15 Quadword Subtract

Format:

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

Operation:

```
Rc ← Rav - Rbv
```

Exceptions:

Integer Overflow

Instruction mnemonics:

SUBQ	Subtract Quadword

Qualifiers:

Integer Overflow Enable (/V)

Description:

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

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

4.4.16 Scaled Quadword Subtract

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

4.5 Logical and Shift Instructions

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

Table 4–6: Logical and Shift Instructions Summary

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

Software Note:

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

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

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

4.5.1 Logical Functions

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

4.5.2 Conditional Move Integer

Format:

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

Operation:

```
IF TEST(Rav, Condition_based_on_Opcode) THEN

  Rc ← Rbv
```

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

Notes:

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

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

is exactly equivalent to:

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

For example, a branchless sequence for:

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

is:

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

4.5.3 Shift Logical

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

SLL	Shift Left Logical
SRL	Shift Right Logical

Qualifiers:

None

Description:

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

4.5.4 Shift Arithmetic

Format:

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

Operation:

```
Rc ← ARITH_RIGHT_SHIFT(Rav, Rbv<5:0>)
```

Exceptions:

None

Instruction mnemonics:

SRA	Shift Right Arithmetic

Qualifiers:

None

Description:

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

4.6 Byte Manipulation Instructions

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

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

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

LDBU and STB are the recommended way to perform byte load and store operations on Alpha implementations that support them; use them rather than the extract, insert, and mask byte instructions described in this section. In particular, the implementation examples in this section that illustrate byte operations are not appropriate for Alpha implementations that support the BWX extension – instead use the recommendations in Section A.4.1.

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

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

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

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

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

4.6.1 Compare Byte

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

CMPBGE	Compare Byte

Qualifiers:

None

Description:

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

Notes:

The result of CMPBGE can be used as an input to ZAP and ZAPNOT.

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

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

To compare two character strings for greater/equal/less:

```
<initialize R1 to aligned QW address of string1>
<initialize R2 to aligned QW address of string2>
LOOP:
      LDQ    R3,  0(R1)         ; Pick up 8 bytes of string1
      LDA    R1,  8(R1)         ; Increment string1 pointer
      LDQ    R4,  0(R2)         ; Pick up 8 bytes of string2
      LDA    R2,  8(R2)         ; Increment string2 pointer
      CMPBGE R31, R3, R6        ; Test for zeros in string1
      XOR    R3,  R4, R5        ; Test for all equal bytes
      BNE    R6,  DONE          ; Exit if a zero found
      BEQ    R5,  LOOP          ; Loop if all equal
DONE: CMPBGE R31, R5, R5        ;
      ...
; At this point, R5 can be used to determine the first not-equal
; byte position (if any), and R6 can be used to determine the
; position of the terminating zero in string1 (if any).
```

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

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

4.6.2 Extract Byte

Format:

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

Operation:

```
CASE
 big_endian_data:  Rbv' ← Rbv XOR 111(2)
 little_endian_data: Rbv' ← Rbv
ENDCASE

CASE
  EXTBL: byte_mask ← 0000 0001(2)
  EXTWx: byte_mask ← 0000 0011(2)
  EXTLx: byte_mask ← 0000 1111(2)
  EXTQx: byte_mask ← 1111 1111(2)
ENDCASE

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

Notes:

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

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

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

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

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

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

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

```
LDQ_U  R1,  X(R11)        ; Ignores va<2:0>, R1 = CBAx xxxx
LDQ_U  R2,  X+3(R11)      ; Ignores va<2:0>, R2 = yyyy yyyD
LDA    R3,  X(R11)        ; R3<2:0> = (X mod 8) = 5
EXTLL  R1,  R3, R1        ; R1 = 0000 0CBA
EXTLH  R2,  R3, R2        ; R2 = 0000 D000
OR     R2,  R1, R1        ; R1 = 0000 DCBA
ADDL   R31, R1, R1        ; R1 = ssss DCBA
```

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

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

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

```
LDQ_U  R1,  X(R11)        ; Ignores va<2:0>, R1 = yBAx xxxx
LDQ_U  R2,  X+1(R11)      ; Ignores va<2:0>, R2 = yBAx xxxx
LDA    R3,  X+1+1(R11)    ; R3<2:0> = 5+1+1 = 7
EXTQL  R1,  R3, R1        ; R1 = 0000 000y
EXTQH  R2,  R3, R2        ; R2 = BAxx xxx0
OR     R2,  R1, R1        ; R1 = BAxx xxxy
SRA    R1, #48, R1        ; R1 = ssss ssBA
```

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

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

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

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

Optimized examples:

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

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

```
LDL    R1, 8(R3)          ; R1 = ssss BAxx
                          ; Faults if R3 is not longword aligned
EXTWL  R1, #2, R1         ; R1 = 0000 00BA
```

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

```
LDL   R1, 8(R3)           ; R1 = ssss BAxx
                          ; Faults if R3 is not longword aligned
SRA   R1, #16, R1         ; R1 = ssss ssBA
```

Big-endian examples:

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

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

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

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

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

4.6.3 Byte Insert

Format:

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

Operation:

```
CASE
 big_endian_data:  Rbv' ← Rbv XOR 111(2)
 little_endian_data: Rbv' ← Rbv
ENDCASE

CASE
  INSBL: byte_mask ← 0000 0000 0000 0001(2)
  INSWx: byte_mask ← 0000 0000 0000 0011(2)
  INSLx: byte_mask ← 0000 0000 0000 1111(2)
  INSQx: byte_mask ← 0000 0000 1111 1111(2)
ENDCASE
byte_mask ← LEFT_SHIFT(byte_mask, Rbv'<2:0>)

CASE
  INSxL:
    byte_loc ← Rbv'<2:0>*8
    temp ← LEFT_SHIFT(Rav, byte_loc<5:0>)
    Rc ← BYTE_ZAP(temp, NOT(byte_mask<7:0>))
  INSxH:
    byte_loc ← 64 - Rbv'<2:0>*8
    temp ← RIGHT_SHIFT(Rav, byte_loc<5:0>)
    Rc ← BYTE_ZAP(temp, NOT(byte_mask<15:8>))
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

4.6.4 Byte Mask

Format:

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

Operation:

```
CASE
 big_endian_data:  Rbv'← Rbv XOR 111(2)
 little_endian_data: Rbv'← Rbv
ENDCASE

CASE
  MSKBL: byte_mask ← 0000 0000 0000 0001(2)
  MSKWx: byte_mask ← 0000 0000 0000 0011(2)
  MSKLx: byte_mask ← 0000 0000 0000 1111(2)
  MSKQx: byte_mask ← 0000 0000 1111 1111(2)
ENDCASE
byte_mask ← LEFT_SHIFT(byte_mask, Rbv'<2:0>)

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

Notes:

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

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

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

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

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

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

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

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

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

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

4.6.5 Sign Extend

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

SEXTB	Sign Extend Byte
SEXTW	Sign Extend Word

Qualifiers:

None

Description:

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

Implementation Note:

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

4.6.6 Zero Bytes

Format:

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

Operation:

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

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

Exceptions:

None

Instruction mnemonics:

ZAP	Zero Bytes
ZAPNOT	Zero Bytes Not

Qualifiers:

None

Description:

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

4.7 Floating-Point Instructions

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

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

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

Note:

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

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

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

4.7.1 Single-Precision Operations

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

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

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

4.7.2 Subsets and Faults

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

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

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

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

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

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

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

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

An implementation that includes IEEE floating-point may implement any subset of the Trap Disable flags (DNOD, DZED, INED, INVD, OVFD, and UNFD) and Denormal Control flags (DNZ and UNDZ) in the FPCR:

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

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

4.7.3 Definitions

The following definitions apply to Alpha floating-point support.

Alpha finite number

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

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

denormal

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

dirty zero

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

infinity

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

LSB

The least significant bit. For a positive finite representable number A, A + 1 LSB is the next larger representative number, and A + ½ LSB is exactly halfway between A and the next larger representable number. For a positive representable number A whose fraction field is not all zeros, A – 1 LSB is the next smaller representable number, and A – ½ LSB is exactly halfway between *A* and the next smaller representable number.

non-finite number

An IEEE infinity, NaN, denormal number, or a VAX dirty zero or reserved operand.

Not-a-Number

An IEEE floating-point bit pattern that represents something other than a number. This comes in two forms: signaling NaNs (for Alpha, those with an initial fraction bit of 0) and quiet NaNs (for Alpha , those with initial fraction bit of 1).

representable result

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

reserved operand

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

trap shadow

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

true result

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

true zero

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

4.7.4 Encodings

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

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

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

Data Format	MIN	MAX
F_floating	2**–127 * 0.5 (0.293873588e–38)	2**127 *(1.0 – 2**–24) (1.7014117e38)
G_floating	2**–1023 * 0.5 (0.5562684646268004e–308)	2**1023 * (1.0 – 2**–53) (0.8988465674311578540 7e308)
S_floating	2**–126 * 1.0 (1.17549435e–38)	2**127 * (2.0 – 2**–23) (3.40282347e38)
T_floating	2**–1022 * 1.0 (2.2250738585072013e–308)	2**1023 * (2.0 – 2**–52) (1.7976931348623158e308)
X_floating	2**–16382*1.0 (See below†)	2**16383*(2.0–2**–112) (See below‡)

† (1.189731495357231765085759326628007 02e4932)

‡ (3.362103143112093506262677817321752 60e–4932)

4.7.5 Rounding Modes

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

VAX Rounding Modes

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

Normal VAX rounding maps the true result to the nearest of two representable results, with true results exactly halfway between mapped to the larger in absolute value (sometimes called biased rounding away from zero); maps true results ≥ MAX + 1/2 LSB in magnitude to an overflow; maps true results < MIN – 1/4 LSB in magnitude to an underflow.

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

IEEE Rounding Modes

For IEEE floating-point operations, four rounding modes are provided: normal rounding (unbiased round to nearest), rounding toward minus infinity, round toward zero, and rounding toward plus infinity. The first three can be specified in the instruction. Rounding toward plus infinity can be obtained by setting the Floating-point Control Register (FPCR) to select it and then specifying dynamic rounding mode in the instruction (See Section 4.7.8). Alpha IEEE arithmetic does rounding before detecting overflow/underflow.

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

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

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

Chopped IEEE rounding maps the true result to the smaller in magnitude of two surrounding representable results; maps true results ≥ MAX + 1 LSB in magnitude to an overflow; and maps non-zero true results < MIN in magnitude to an underflow.

Dynamic rounding mode uses the IEEE rounding mode selected by the FPCR register and is described in more detail in Section 4.7.8.

The following tables summarize the floating-point rounding modes:

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

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

4.7.6 Computational Models

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

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

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

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

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

4.7.6.1 VAX-Format Arithmetic with Precise Exceptions

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

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

4.7.6.2 High-Performance VAX-Format Arithmetic

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

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

4.7.6.3 IEEE-Compliant Arithmetic

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

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

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

4.7.6.4 IEEE-Compliant Arithmetic Without Inexact Exception

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

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

4.7.6.5 High-Performance IEEE-Format Arithmetic

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

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

4.7.7 Trapping Modes

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

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

4.7.7.1 VAX Trapping Modes

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

When no trap mode is specified (the default):

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

When /U or /V mode is specified:

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

When /S mode is specified:

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

When /SU or /SV mode is specified:

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

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

Table 4–8: VAX Trapping Modes Summary

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

4.7.7.2 IEEE Trapping Modes

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

When no trap mode is specified (the default):

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

When /U or /V mode is specified :

- Arithmetic is performed on IEEE finite numbers.
- Operations give imprecise traps whenever the following occur:
  - an operand is a non-finite number
  - an underflow
  - an integer overflow
  - a floating overflow
  - a divide-by-zero
  - an invalid operation

- Traps are imprecise, and it is not always possible to determine which instruction triggered a trap or the operands of that instruction.
- An underflow trap produces a zero.
- A conversion to integer trap with an integer overflow produces the low-order bits of the integer.
- The result of any other operation that traps is UNPREDICTABLE.

When /SU or /SV mode is specified:

- Arithmetic is performed on all IEEE values, both finite and non-finite.
- Alpha systems support all IEEE features except inexact exception (which requires /SUI or /SVI):
  - The IEEE standard specifies a default where exceptions do not fault or trap.In combination with the FPCR, this mode allows disabling exceptions and producing IEEE compliant nontrapping results. See Sections 4.7.7.10 and 4.7.7.11.
  - Each Alpha operating system provides a way to optionally signal IEEE floating-point exceptions. This mode enables the IEEE status flags that keep a record of each exception that is encountered. An Alpha operating system uses the IEEE floating-point control (FP_C) quadword, described in Section B.2.1, to maintain the IEEE status flags and to enable calls to IEEE user signal handlers.
- Exceptions signaled in this mode are precise and an application can locate the instruction that caused the exception, along with its operand values. See Section 4.7.7.3.

When /SUI or /SVI mode is specified:

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

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

Table 4–9: Summary of IEEE Trapping Modes

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

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

Trap Mode	Notation	Meaning
Integer overflow enabled and inexact disabled	/V /SV	Imprecise Precise exception completion
Integer overflow enabled and inexact enabled	/SVI	Precise exception completion

4.7.7.3 Arithmetic Trap Completion

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

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

See Section B.2 for more information.

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

- If AMASK returns with bit 9 clear, floating-point traps are imprecise. Exception completion requires that generated code must obey the trap shadow rules in Section 4.7.7.3.1, with a trap shadow length as described in Section 4.7.7.3.2.
- If AMASK returns with bit 9 set, the hardware implements precise floating-point traps. If the instruction has any valid qualifier combination that includes /S, the trap PC points to the instruction that immediately follows the instruction that triggered the trap. The trap shadow contains zero instructions; exception completion does not require that the generated code follow the conditions in Section 4.7.7.3.1 and the length rules in Section 4.7.7.3.2.

4.7.7.3.1 Trap Shadow Rules

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

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

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

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

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

Conditions:

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

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

4.7.7.3.2 Trap Shadow Length Rules

The trap shadow length rules in Table 4–10 apply only to those floating-point instructions with any valid qualifier combination that includes a /S trap qualifier. Further, the instruction to which the trap shadow extends is not part of the trap shadow and that instruction is not executed prior to the arithmetic trap that is signaled by the trigger instruction.

Implementation notes:

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

Table 4–10: Trap Shadow Length Rules

Floating-Point Instruction Group	Trap Shadow Extends Until Any of the Following Occurs:
Floating-point operate (except DIVx and SQRTx)	• Encountering a CALL_PAL, EXCB, or TRAPB instruction. • The result is consumed by any instruction except floating-point STx. • The fourth instruction† after the result is consumed by a floating-point STx instruction. Or, following the floating-point STx of the result, the result of a LDx that loads the stored value is consumed by any instruction. • The result of a subsequent floating-point operate instruction is consumed by any instruction except floating-point STx. • The second instruction† after the result of a subsequent floating-point operate instruction is consumed by a floating-point STx instruction. • The result of a subsequent floating-point DIVx or SQRTx instruction is consumed by any instruction.
Floating-point DIVx	• Encountering a CALL_PAL, EXCB, or TRAPB instruction. • The result is consumed by any instruction except floating-point STx. • The fourth instruction† after the result is consumed by a floating-point STx instruction. Or, following the floating-point STx of the result, the result of a LDx that loads the stored value is consumed by any instruction. • The result of a subsequent floating-point DIVx is consumed by any instruction.

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

Floating-Point Instruction Group	Trap Shadow Extends Until Any of the Following Occurs:
Floating-point SQRTx	• Encountering a CALL_PAL, EXCB, or TRAPB instruction. • The result is consumed by any instruction. • The result of a subsequent SQRTx instruction is consumed by any instruction.

† The length of four instructions is a conservative estimate of how far the trap shadow may extend past a consuming floating-point STx instruction. The length of two instructions is a conservative estimate of how far the trap shadow may extend after a subsequent floating-point operate instruction is consumed by a floating-point STx instruction. Compilers can make a more precise estimate by consulting the *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors Hardware Reference Manual, EC-QD2RA-TE*.

4.7.7.4 Invalid Operation (INV) Arithmetic Trap

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

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

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

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

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

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

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

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

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

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

4.7.7.5 Division by Zero (DZE) Arithmetic Trap

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

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

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

4.7.7.6 Overflow (OVF) Arithmetic Trap

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

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

4.7.7.7 Underflow (UNF) Arithmetic Trap

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

If an underflow occurs, a true zero (64 bits of zero) is always stored in the result register. In the case of an IEEE operation that takes an underflow arithmetic trap, a true zero is stored even if the result after rounding would have been –0 (underflow below the negative denormal range).

If an underflow occurs and underflow traps are enabled by the instruction, an underflow arithmetic trap is signaled. However, under some conditions, the FPCR can dynamically disable the trap, as described in Section 4.7.7.10, producing the result described in Section 4.7.10, as modified by the UNDZ bit described in Section 4.7.7.11.

4.7.7.8 Inexact Result (INE) Arithmetic Trap

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

If an inexact result occurs, the normal rounded result is still stored in the result register. If an inexact result occurs and inexact result traps are enabled by the instruction, an inexact result arithmetic trap is signaled. However, under some conditions, the FPCR can dynamically disable the trap; see Section 4.7.7.10 for information.

4.7.7.9 Integer Overflow (IOV) Arithmetic Trap

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

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

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

4.7.7.10 IEEE Floating-Point Trap Disable Bits

In the case of IEEE exception completion modes, any of the traps described in Sections 4.7.7.4 through 4.7.7.9 may be disabled by setting the appropriate trap disable bit in the FPCR. The trap disable bits only affect the IEEE trap modes when the instruction is modified by any valid qualifier combination that includes the /S (exception completion) qualifier. The trap disable bits (DNOD, DZED, INED, INVD, OVFD, and UNFD) do not affect any of the VAX trap modes.

If a trap disable bit is set and the corresponding trap condition occurs, the hardware implementation sets the result of the operation to the nontrapping result value as specified in the IEEE standard and Section 4.7.10 and modified by the denormal control bits. If the implementation is unable to calculate the required result, it ignores the trap disable bit and signals a trap as usual.

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

4.7.7.11 IEEE Denormal Control Bits

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

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

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

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

- An IEEE operation with a denormal operand never generates an overflow, underflow, or inexact result arithmetic trap.
- Dividing by a denormal operand is a division by zero or invalid operation as appropriate.
- Multiplying a denormal by infinity is an invalid operation.
- A SQRT of a negative denormal produces a –0 instead of an invalid operation.
- A denormal operand, treated as zero, does not take the denormal operand exception trap controlled by the DNOD bit in the FPCR.

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

4.7.8 Floating-Point Control Register (FPCR)

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

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

Floating-point operates that belong to the IEEE subset and CVTQL, which belongs to both

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

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

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

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

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

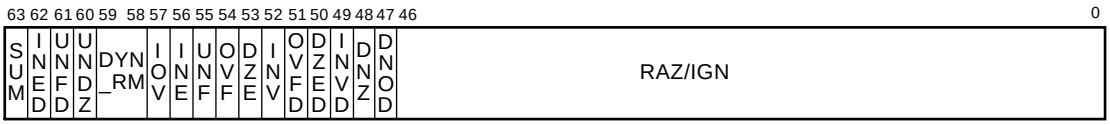

Table 4–11: Floating-Point Control Register (FPCR) Bit Descriptions

Bit	Description (Meaning When Set)
63	Summary Bit (SUM). Records bitwise OR of FPCR exception bits. Equal to FPCR<57 |56 | 55 | 54 | 53 | 52>.
62	Inexact Disable (INED)†. Suppress INE trap and place correct IEEE nontrapping result in the destination register.
61	Underflow Disable (UNFD)†. Suppress UNF trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping result. The correct result value is determined according to the value of the UNDZ bit.
60	Underflow to Zero (UNDZ)†. When set together with UNFD, on underflow, the hardware places a true zero (64 bits of zero) in the destination register rather than the result specified by the IEEE standard.
59–58	Dynamic Rounding Mode (DYN). Indicates the rounding mode to be used by an IEEE floating-point operate instruction when the instruction's function field specifies dynamic mode (/D). Assignments are:

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

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

Bit	Description (Meaning When Set)
57	Integer Overflow (IOV). An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision.
56	Inexact Result (INE). A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.
55	Underflow (UNF). A floating arithmetic or conversion operation underflowed the destination exponent.
54	Overflow (OVF). A floating arithmetic or conversion operation overflowed the destination exponent.
53	Division by Zero (DZE). An attempt was made to perform a floating divide operation with a divisor of zero.
52	Invalid Operation (INV). An attempt was made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values were illegal.
51	Overflow Disable (OVFD)†. Suppress OVF trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping results.
50	Division by Zero Disable (DZED)†. Suppress DZE trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping results.
49	Invalid Operation Disable (INVD)†. Suppress INV trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping results.
48	Denormal Operands to Zero (DNZ)†. Treat all denormal operands as a signed zero value with the same sign as the denormal.
47	Denormal Operand Exception Disable (DNOD)†. Suppress INV trap for valid operations that involve denormal operand values and place the correct IEEE nontrapping result in the destination register if the implementation is capable of processing the denormal operand. If the result of the operation underflows, the correct result is determined according to the value of the UNDZ bit. If DNZ is set, DNOD has no effect because a denormal operand is treated as having a zero value instead of a denormal value.
46–0	Reserved. Read as Zero. Ignored when written.

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

FPCR is read from and written to the floating-point registers by the MT_FPCR and MF_FPCR instructions respectively, which are described in Section 4.7.8.1.

FPCR and the instructions to access it are required for an implementation that supports floating-point (see Section 4.7.8). On implementations that do not support floating-point, the instructions that access FPCR (MF_FPCR and MT_FPCR) take an Illegal Instruction Trap.

Software Note:

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

4.7.8.1 Accessing the FPCR

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

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

Consider the following example:

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

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

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

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

4.7.8.2 Default Values of the FPCR

Processor initialization leaves the value of FPCR UNPREDICTABLE.

Software Note:

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

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

4.7.8.3 Saving and Restoring the FPCR

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

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

Software Notes:

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

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

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

   Trap interface software (usually the operating system) keeps a set of software status bits and a mask of the traps that the user wants to receive. Code is generated with the /SUI qualifiers. For a particular exception, the software clears the corresponding trap disable bit if either the corresponding software status bit is 0 or if the user wants to receive such traps. If a trap occurs, the software locates the offending instruction in the trap shadow, simulates it and sets any of the software status bits that are appropriate. Then, the software either delivers the trap to the user program or disables further delivery of such traps. The user program must interface to this trap interface software to set or clear any of the software status bits or to enable or disable floating-point traps. See Section B.2.

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

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

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

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

4.7.9 Floating-Point Instruction Function Field Format

The function code for IEEE and VAX floating-point instructions, bits <15..5>, contain the function field. That field is shown in Figure 4–2 and described for IEEE floating-point in Table 4–12 and for VAX floating-point in Table 4–13. Function codes for the independent floating-point instructions, those with opcode 17_{16}, do not correspond to the function fields below.

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

Figure 4–2: Floating-Point Instruction Function Field

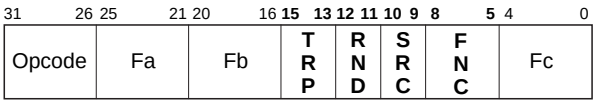

Table 4–12: IEEE Floating-Point Function Field Bit Summary

Bits	Field	Meaning†
15–13	TRP	Trapping modes:

Contents	Meaning for Opcodes 14_{16} and 16_{16}
000	Imprecise (default)
001	Underflow enable (/U) — floating-point output Integer overflow enable (/V) — integer output
010	UNPREDICTABLE for opcode 16_{16} instructions Reserved for opcode 14_{16} instructions
011	UNPREDICTABLE for opcode 16_{16} instructions Reserved for opcode 14_{16} instructions
100	UNPREDICTABLE for opcode 16_{16} instructions Reserved for opcode 14_{16} instructions
101	/SU — floating-point output /SV — integer output
110	UNPREDICTABLE for opcode 16_{16} instructions Reserved for opcode 14_{16} instructions
111	/SUI — floating-point output /SVI — integer output

12–11 RND Rounding modes:

Contents	Meaning for Opcodes 16_{16} and 14_{16}
00	Chopped (/C)
01	Minus infinity (/M)
10	Normal (default)
11	Dynamic (/D)

10–9 SRC Source datatype:

Contents	Meaning for Opcode 16_{16}	Meaning for Opcode 14_{16}
00	S_floating	S_floating
01	Reserved	Reserved
10	T_floating	T_floating
11	Q_fixed	Reserved

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

Bits	Field	Meaning†		
8–5	FNC	Instruction class:		
		Contents	**Meaning for Opcode 16_{16}**	**Meaning for Opcode 14_{16}**
		0000	ADDx	Reserved
		0001	SUBx	Reserved
		0010	MULx	Reserved
		0011	DIVx	Reserved
		0100	CMPxUN	ITOFS/ITOFT
		0101	CMPxEQ	Reserved
		0110	CMPxLT	Reserved
		0111	CMPxLE	Reserved
		1000	Reserved	Reserved
		1001	Reserved	Reserved
		1010	Reserved	Reserved
		1011	Reserved	SQRTS/SQRTT
		1100	CVTxS	Reserved
		1101	Reserved	Reserved
		1110	CVTxT	Reserved
		1111	CVTxQ	Reserved

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

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

Bits	Field	Meaning
15–13	TRP	Trapping modes:

Contents	Meaning for Opcodes 14_{16} and 15_{16}
000	Imprecise (default)
001	Underflow enable (/U) – floating-point output Integer overflow enable (/V) – integer output
010	UNPREDICTABLE for opcode 15_{16} instructions Reserved for opcode 14_{16} instructions
011	UNPREDICTABLE for opcode 15_{16} instructions Reserved for opcode 14_{16} instructions
100	/S – Exception completion enable
101	/SU – floating-point output /SV – integer output
110	UNPREDICTABLE for opcode 15_{16} instructions Reserved for opcode 14_{16} instructions
111	UNPREDICTABLE for opcode 15_{16} instructions Reserved for opcode 14_{16} instructions

12–11	RND	Rounding modes:

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

10–9	SRC	Source datatype:†

Contents	Meaning for Opcode 15_{16}	Meaning for Opcode 14_{16}
00	F_floating	F_floating
01	D_floating	F_floating
10	G_floating	G_floating
11	Q_fixed	Reserved

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

Bits	Field	Meaning		
8–5	FNC	Instruction class:		
		Contents	**Meaning for Opcode 15_{16}**	**Meaning for Opcode 14_{16}**
		0000	ADDx	Reserved
		0001	SUBx	Reserved
		0010	MULx	Reserved
		0011	DIVx	Reserved
		0100	CMPxUN	ITOFF
		0101	CMPxEQ	Reserved
		0110	CMPxLT	Reserved
		0111	CMPxLE	Reserved
		1000	Reserved	Reserved
		1001	Reserved	Reserved
		1010	Reserved	SQRTF/SQRTG
		1011	Reserved	Reserved
		1100	CVTxF	Reserved
		1101	CVTxD	Reserved
		1110	CVTxG	Reserved
		1111	CVTxQ	Reserved

† In the SRC field, both 00 and 01 specify the F_floating source datatype for opcode 14_{16}.

4.7.10 IEEE Standard

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

This standard leaves certain operations as implementation dependent. The remainder of this section specifies the behavior of the Alpha architecture in these situations. Note that this behavior may be supplied by either hardware (if the invalid operation disable, or INVD, bit is implemented) or by software. See Sections 4.7.7.10, 4.7.7.11, 4.7.8, 4.7.8.3, and Section B.1.

4.7.10.1 Conversion of NaN and Infinity Values

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

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

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

4.7.10.2 Copying NaN Values

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

4.7.10.3 Generating NaN Values

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

4.7.10.4 Propagating NaN Values

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

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

4.8 Memory Format Floating-Point Instructions

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

The instructions are summarized in Table 4–14.

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

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

4.8.1 Load F_floating

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

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

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

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDF Load F_floating

Qualifiers:

None

Description:

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

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

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

4.8.2 Load G_floating

Format:

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

Operation:

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

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDG	Load G_floating (Load D_floating)

Qualifiers:

None

Description:

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

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

4.8.3 Load S_floating

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

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

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

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDS	Load S_floating (Load Longword Integer)

Qualifiers:

None

Description:

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

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

4.8.4 Load T_floating

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

Fa ← (va)<63:0>
```

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDT	Load T_floating (Load Quadword Integer)

Qualifiers:

None

Description:

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

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

4.8.5 Store F_floating

Format:

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

Operation:

```
va ← {Rbv + SEXT(disp)}

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

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

Exceptions:

Access Violation

Fault on Write

Alignment

Translation Not Valid

Instruction mnemonics:

STF	Store F_floating

Qualifiers:

None

Description:

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

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

4.8.6 Store G_floating

Format:

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

Operation:

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

Exceptions:

Access Violation
Fault on Write
Alignment
Translation Not Valid

Instruction mnemonics:

STG	Store G_floating (Store D_floating)

Qualifiers:

None

Description:

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

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

4.8.7 Store S_floating

Format:

STS	Fa.rs,disp.ab(Rb.ab)	!Memory format

Operation:

```
va ← {Rbv + SEXT(disp)}

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

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

Exceptions:

Access Violation
Fault on Write
Alignment
Translation Not Valid

Instruction mnemonics:

STS	Store S_floating (Store Longword Integer)

Qualifiers:

None

Description:

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

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

4.8.8 Store T_floating

Format:

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

Operation:

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

Exceptions:

Access Violation
Fault on Write
Alignment
Translation Not Valid

Instruction mnemonics:

STT Store T_floating (Store Quadword Integer)

Qualifiers:

None

Description:

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

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

4.9 Branch Format Floating-Point Instructions

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

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

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

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

Table 4–15: Floating-Point Branch Instructions Summary

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

4.9.1 Conditional Branch

Format:

FBxx	Fa.rq,disp.al	!Branch format

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

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

Notes:

- To branch properly on non-finite operands, compare to F31, then branch on the result of the compare.
- The largest negative integer ($8000\ 0000\ 0000\ 0000_{16}$) is the same bit pattern as floating minus zero, so it is treated as equal to zero by the branch instructions. To branch properly on the largest negative integer, convert it to floating or move it to an integer register and do an integer branch.

4.10 Floating-Point Operate Format Instructions

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

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

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

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

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

The floating-point operate instructions are summarized in Table 4–16.

Table 4–16: Floating-Point Operate Instructions Summary

Mnemonic	Operation	Subset
Bit and FPCR Operations:		
CPYS	Copy Sign	Both
CPYSE	Copy Sign and Exponent	Both
CPYSN	Copy Sign Negate	Both
CVTLQ	Convert Longword to Quadword	Both
CVTQL	Convert Quadword to Longword	Both
FCMOVxx	Floating Conditional Move	Both
MF_FPCR	Move from Floating-point Control Register	Both
MT_FPCR	Move to Floating-point Control Register	Both

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

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

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

Mnemonic	Operation	Subset
Arithmetic Operations		
MULF	Multiply F_floating	VAX
MULG	Multiply G_floating	VAX
MULS	Multiply S_floating	IEEE
MULT	Multiply T_floating	IEEE
SQRTF	Square root F_floating	VAX
SQRTG	Square root G_floating	VAX
SQRTS	Square root S_floating	IEEE
SQRTT	Square root T_floating	IEEE
SUBF	Subtract F_floating	VAX
SUBG	Subtract G_floating	VAX
SUBS	Subtract S_floating	IEEE
SUBT	Subtract T_floating	IEEE

4.10.1 Copy Sign

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

No checking of the operands is performed.

Notes:

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

4.10.2 Convert Integer to Integer

Format:

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

Operation:

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

Exceptions:

Integer Overflow, CVTQL only

Instruction mnemonics:

CVTLQ	Convert Longword to Quadword
CVTQL	Convert Quadword to Longword

Qualifiers:

Trapping:	Exception Completion (/S) (CVTQL only)
	Integer Overflow Enable (/V) (CVTQL only)

Description:

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

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

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

4.10.3 Floating-Point Conditional Move

Format:

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

Operation:

```
IF TEST(Fav, Condition_based_on_Opcode) THEN

  Fc ← Fbv
```

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

Notes:

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

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

is exactly equivalent to:

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

For example, a branchless sequence for:

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

is:

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

4.10.4 Move from/to Floating-Point Control Register

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

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

Qualifiers:

None

Description:

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

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

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

4.10.5 VAX Floating Add

Format:

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

Operation:

Fc ← Fav + Fbv

Exceptions:

Invalid Operation
Overflow
Underflow

Instruction mnemonics:

ADDF	Add F_floating
ADDG	Add G_floating

Qualifiers:

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

Description:

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

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

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs. See Section 4.7.7 for details of the stored result on overflow or underflow.

4.10.6 IEEE Floating Add

Format:

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

Operation:

Fc ← Fav + Fbv

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

ADDS	Add S_floating
ADDT	Add T_floating

Qualifiers:

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

Description:

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

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

See Section 4.7.7 for details of the stored result on overflow, underflow, or inexact result.

4.10.7 VAX Floating Compare

Format:

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

Operation:

```
IF Fav SIGNED_RELATION Fbv THEN
  Fc ← 4000 0000 0000 0000(16)
ELSE
  Fc ← 0000 0000 0000 0000(16)
```

Exceptions:

Invalid Operation

Instruction mnemonics:

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

Qualifiers:

Trapping:	Exception Completion (/S)

Description:

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

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

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

Notes:

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

4.10.8 IEEE Floating Compare

Format:

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

Operation:

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

Exceptions:

Invalid Operation

Instruction mnemonics:

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

Qualifiers:

Trapping:	Exception Completion (/SU)

Description:

The two operands in Fa and Fb are compared. If the relationship specified by the qualifier is true, a non-zero floating value (2.0) is written to register Fc; otherwise, a true zero is written to Fc.

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

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

Notes:

- In order to use CMPTxx with exception completion handling, it is necessary to specify the /SU IEEE trap mode, even though an underflow trap is not possible.
- Compare Less Than A,B is the same as Compare Greater Than B,A; Compare Less Than or Equal A,B is the same as Compare Greater Than or Equal B,A. Therefore, only the less-than operations are included.

4.10.9 Convert VAX Floating to Integer

Format:

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

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

Invalid Operation
Integer Overflow

Instruction mnemonics:

CVTGQ	Convert G_floating to Quadword

Qualifiers:

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

Description:

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

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

See Section 4.7.7 for details of the stored result on integer overflow.

4.10.10 Convert Integer to VAX Floating

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

CVTQF	Convert Quadword to F_floating
CVTQG	Convert Quadword to G_floating

Qualifiers:

Rounding:	Chopped (/C)

Description:

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

4.10.11 Convert VAX Floating to VAX Floating

Format:

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

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

Invalid Operation
Overflow
Underflow

Instruction mnemonics:

CVTDG	Convert D_floating to G_floating
CVTGD	Convert G_floating to D_floating
CVTGF	Convert G_floating to F_floating

Qualifiers:

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

Description:

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

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

See Section 4.7.7 for details of the stored result on overflow or underflow.

Notes:

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

4.10.12 Convert IEEE Floating to Integer

Format:

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

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

Invalid Operation
Inexact Result
Integer Overflow

Instruction mnemonics:

CVTTQ	Convert T_floating to Quadword

Qualifiers:

Rounding:	Dynamic (/D)
	Minus infinity (/M)
	Chopped (/C)
Trapping:	Exception Completion (/S)
	Integer Overflow Enable (/V)
	Inexact Enable (/I)

Description:

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

See Section 4.7.7 for details of the stored result on integer overflow and inexact result.

4.10.13 Convert Integer to IEEE Floating

Format:

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

Operation:

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

Exceptions:

Inexact Result

Instruction mnemonics:

CVTQS	Convert Quadword to S_floating
CVTQT	Convert Quadword to T_floating

Qualifiers:

Rounding:	Dynamic (/D)
	Minus infinity (/M)
	Chopped (/C)
Trapping:	Exception Completion (/S)
	Inexact Enable (/I)

Description:

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

See Section 4.7.7 for details of the stored result on inexact result.

Notes:

- In order to use CVTQS or CVTQT with exception completion handling, it is necessary to specify the /SUI IEEE trap mode, even though an underflow trap is not possible.

4.10.14 Convert IEEE S_Floating to IEEE T_Floating

Format:

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

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

Invalid Operation

Instruction mnemonics:

CVTST	Convert S_floating to T_floating

Qualifiers:

Trapping:	Exception Completion (/S)

Description:

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

Notes:

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

4.10.15 Convert IEEE T_Floating to IEEE S_Floating

Format:

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

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

CVTTS	Convert T_floating to S_floating

Qualifiers:

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

Description:

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

See Section 4.7.7 for details of the stored result on overflow, underflow, or inexact result.

4.10.16 VAX Floating Divide

Format:

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

Operation:

```
Fc ← Fav / Fbv
```

Exceptions:

Invalid Operation
Division by Zero
Overflow
Underflow

Instruction mnemonics:

DIVF	Divide F_floating
DIVG	Divide G_floating

Qualifiers:

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

Description:

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

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

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

A division by zero trap is signaled if Fbv is zero. The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.7 for details of the stored result on overflow or underflow.

4.10.17 IEEE Floating Divide

Format:

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

Operation:

Fc ← Fav / Fbv

Exceptions:

Invalid Operation
Division by Zero
Overflow
Underflow
Inexact Result

Instruction mnemonics:

DIVS	Divide S_floating
DIVT	Divide T_floating

Qualifiers:

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

Description:

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

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

See Section 4.7.7 for details of the stored result on overflow, underflow, or inexact result.

4.10.18 Floating-Point Register to Integer Register Move

Format:

FTOIx	Fa.rq,Rc.wq	!Floating-point Operate format

Operation:

```
CASE:
   FTOIS:
     Rc<63:32> ← SEXT(Fav<63>)
     Rc<31:0> ← Fav<63:62> || Fav <58:29>
   FTOIT:
     Rc <- Fav
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

FTOIS	Floating-point to Integer Register Move, S_floating
FTOIT	Floating-point to Integer Register Move, T_floating

Qualifiers:

None

Description:

Data in a floating-point register file is moved to an integer register file.

The Fb field must be F31.

The instructions do not interpret bits in the register files; specifically, the instructions do not trap on non-finite values. Also, the instructions do not access memory.

FTOIS is exactly equivalent to the sequence:

```
STS
LDL
```

FTOIT is exactly equivalent to the sequence:

```
STT
LDQ
```

Software Note:

FTOIS and FTOIT are no slower than the corresponding store/load sequence and can be significantly faster.

4.10.19 Integer Register to Floating-Point Register Move

Format:

ITOFx	Ra.rq,Fc.wq	!Floating-point Operate format

Operation:

```
CASE:
  ITOFF:
    Fc ← Rav<31> || MAP_F(Rav<30:23> || Rav<22:0> || 0<28:0>
  ITOFS:
    Fc ← Rav<31> || MAP_S(Rav<30:23> || Rav<22:0> || 0<28:0>
  ITOFT:
    Fc <- Rav
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

ITOFF	Integer to Floating-point Register Move, F_floating
ITOFS	Integer to Floating-point Register Move, S_floating
ITOFT	Integer to Floating-point Register Move, T_floating

Qualifiers:

None

Description:

Data in an integer register file is moved to a floating-point register file.

The Rb field must be R31.

The instructions do not interpret bits in the register files; specifically, the instructions do not trap on non-finite values. Also, the instructions do not access memory.

ITOFF is equivalent to the following sequence, except that the word swapping that LDF normally performs is not performed by ITOFF:

```
STL
LDF
```

ITOFS is exactly equivalent to the sequence:

```
STL
LDS
```

ITOFT is exactly equivalent to the sequence:

```
STQ
LDT
```

Software Note:

ITOFF, ITOFS, and ITOFT are no slower than the corresponding store/load sequence and can be significantly faster.

4.10.20 VAX Floating Multiply

Format:

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

Operation:

Fc ← Fav * Fbv

Exceptions:

Invalid Operation
Overflow
Underflow

Instruction mnemonics:

MULF	Multiply F_floating
MULG	Multiply G_floating

Qualifiers:

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

Description:

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

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

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

See Section 4.7.7 for details of the stored result on overflow or underflow.

4.10.21 IEEE Floating Multiply

Format:

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

Operation:

Fc ← Fav * Fbv

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

MULS	Multiply S_floating
MULT	Multiply T_floating

Qualifiers:

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

Description:

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

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

See Section 4.7.7 for details of the stored result on overflow, underflow, or inexact result.

4.10.22 VAX Floating Square Root

Format:

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

Operation:

```
Fc ← Fb ** (1/2)
```

Exceptions:

Invalid operation

Instruction mnemonics:

SQRTF	Square root F_floating
SQRTG	Square root G_floating

Qualifiers:

Rounding:	Chopped (/C)
Trapping:	Exception Completion (/S)
	Underflow Enable (/U) — See Notes below

Description:

The square root of the floating-point operand in register Fb is written to register Fc. (The Fa field of this instruction must be set to a value of F31.)

The result is rounded or chopped to the specified precision. The single-precision operation on a canonical single-precision value produces a canonical single-precision result.

An invalid operation is signaled if the operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). An invalid operation is signaled if the sign of the operand is negative.

The contents of the Fc are UNPREDICTABLE if an invalid operation is signaled.

Notes:

- Floating-point overflow and underflow are not possible for square root operation. The underflow enable qualifier is ignored.

4.10.23 IEEE Floating Square Root

Format:

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

Operation:

```
Fc ← Fb ** (1/2)
```

Exceptions:

Inexact result
Invalid operation

Instruction mnemonics:

SQRTS	Square root S_floating
SQRTT	Square root T_floating

Qualifiers:

Rounding:	Chopped (/C)
	Dynamic (/D)
	Minus infinity (/M)
Trapping:	Inexact Enable (/I)
	Exception Completion (/S)
	Underflow Enable (/U) — See Notes below

Description:

The square root of the floating-point operand in register Fb is written to register Fc. (The Fa field of this instruction must be set to a value of F31.)

The result is rounded to the specified precision. The single-precision operation on a canonical single-precision value produces a canonical single-precision result.

An invalid operation is signaled if the sign of the operand is less than zero. However, SQRT (–0) produces a result of –0.

Notes:

- Floating-point overflow and underflow are not possible for square root operation. The underflow enable qualifier is ignored.

4.10.24 VAX Floating Subtract

Format:

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

Operation:

Fc ← Fav - Fbv

Exceptions:

Invalid Operation
Overflow
Underflow

Instruction mnemonics:

SUBF	Subtract F_floating
SUBG	Subtract G_floating

Qualifiers:

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

Description:

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

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

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

See Section 4.7.7 for details of the stored result on overflow or underflow.

4.10.25 IEEE Floating Subtract

Format:

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

Operation:

Fc ← Fav - Fbv

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

SUBS	Subtract S_floating
SUBT	Subtract T_floating

Qualifiers:

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

Description:

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

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

See Section 4.7.7 for details of the stored result on overflow, underflow, or inexact result.

4.11 Miscellaneous Instructions

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

Table 4–17: Miscellaneous Instructions Summary

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

4.11.1 Architecture Mask

Format:

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

Operation:

```
Rc ← Rbv AND {NOT CPU_feature_mask}
```

Exceptions:

None

Instruction mnemonics:

AMASK	Architecture Mask

Qualifiers:

None

Description:

Rbv represents a mask of the requested architectural extensions. Bits are cleared that correspond to architectural extensions that are present. Reserved bits and bits that correspond to absent extensions are copied unchanged. In either case, the result is placed in Rc. If the result is zero, all requested features are present.

Software may specify an Rbv of all 1's to determine the complete set of architectural extensions implemented by a processor. Assigned bit definitions are located in Section D.3.

Ra must be R31 or the result in Rc is UNPREDICTABLE and it is UNPREDICTABLE whether an exception is signaled.

Software Note:

Use this instruction to make instruction-set decisions; use IMPLVER to make code-tuning decisions.

Implementation Note:

Instruction encoding is implemented as follows:

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

- On 21164A (EV56), 21164PC (PCA56), and 21264 (EV6), AMASK correctly indicates support for architecture extensions by copying Rbv to Rc and clearing appropriate bits.

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

4.11.2 Call Privileged Architecture Library

Format:

CALL_PAL	fnc.ir	!PAL format

Operation:

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

Exceptions:

None

Instruction mnemonics:

CALL_PAL	Call Privileged Architecture Library

Qualifiers:

None

Description:

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

4.11.3 Evict Data Cache Block

Format:

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

Operation:

```
va ← Rbv

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

Exceptions:

None

Instruction mnemonics:

ECB	Evict Cache Block

Qualifiers:

None

Description:

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

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

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

Software Note:

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

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

Implementation Note:

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

4.11.4 Exception Barrier

Format:

```
EXCB                                                    ! Memory format
```

Operation:

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

Exceptions:

None

Instruction mnemonics:

EXCB	Exception Barrier

Qualifiers:

None

Description:

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

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

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

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

4.11.5 Prefetch Data

Format:

FETCHx	0(Rb.ab)	!Memory format

Operation:

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

Exceptions:

None

Instruction mnemonics:

FETCH	Prefetch Data
FETCH_M	Prefetch Data, Modify Intent

Qualifiers:

None

Description:

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

Implementation Note:

FETCHx is intended to help software overlap memory latencies when such latencies are on the order of at least 100 cycles. FETCHx is unlikely to help (or be implemented) for significantly shorter memory latencies. Code scheduling and cache-line prefetching (See Section A.3.5) should be used to overlap such shorter latencies.

Existing Alpha implementations (through the 21264) have memory latencies that are too short to profitably implement FETCHx. Therefore, FETCHx does not improve memory performance in existing Alpha implementations.

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

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

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

Implementation Note:

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

4.11.6 Implementation Version

Format:

```
IMPLVER     Rc                                  !Operate format
```

Operation:

```
Rc ← value, which is defined in Appendix D
```

Exceptions:

None

Instruction mnemonics:

IMPLVER	Implementation Version

Description:

A small integer is placed in Rc that specifies the major implementation version of the processor on which it is executed. This information can be used to make code-scheduling or tuning decisions, or the information can be used to branch to different pieces of code optimized for different implementations.

Notes:

- The value returned by IMPLVER does not identify the particular processor type. Rather, it identifies a group of processors that can be treated similarly for performance characteristics such as scheduling. Ra must be R31 and Rb must be the literal #1 or the result in Rc is UNPREDICTABLE and it is UNPREDICTABLE whether an exception is signaled.

Software Note:

Use this instruction to make code-tuning decisions; use AMASK to make instruction-set decisions.

4.11.7 Memory Barrier

Format:

```
MB                                                    !Memory format
```

Operation:

```
{Guarantee that all subsequent loads or stores
will not access memory until after all previous
loads and stores have accessed memory, as
observed by other processors.}
```

Exceptions:

None

Instruction mnemonics:

MB	Memory Barrier

Qualifiers:

None

Description:

The use of the Memory Barrier (MB) instruction is required only in multiprocessor systems.

In the absence of an MB instruction, loads and stores to different physical locations are allowed to complete out of order on the issuing processor as observed by other processors. The MB instruction allows memory accesses to be serialized on the issuing processor as observed by other processors. See Chapter 5 for details on using the MB instruction to serialize these accesses. Chapter 5 also details coordinating memory accesses across processors.

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

4.11.8 Read Processor Cycle Counter

Format:

RPCC	Ra.wq	!Memory format

Operation:

```
Ra ← {cycle counter}
```

Exceptions:

None

Instruction mnemonics:

RPCC	Read Processor Cycle Counter

Qualifiers:

None

Description:

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

See Section 3.1.5 for a description of the PCC.

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

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

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

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

4.11.9 Trap Barrier

Format:

```
TRAPB                                                    !Memory format
```

Operation:

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

Exceptions:

None

Instruction mnemonics:

TRAPB	Trap Barrier

Qualifiers:

None

Description:

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

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

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

Use of TRAPB should be compared with use of the EXCB instruction; see Section 4.11.4.

4.11.10 Write Hint

Format:

WH64	(Rb.ab)	! Memory format

Operation:

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

Exceptions:

None

Instruction mnemonics:

WH64	Write Hint - 64 Bytes

Qualifiers:

None

Description:

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

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

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

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

Software Note:

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

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

Implementation Note:

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

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

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

4.11.11 Write Memory Barrier

Format:

WMB !Memory format

Operation:

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

Exceptions:

None

Instruction mnemonics:

WMB Write Memory Barrier

Qualifiers:

None

Description:

The WMB instruction provides a way for software to control write buffers. It guarantees that writes preceding the WMB are not aggregated with writes that follow the WMB.

WMB guarantees that writes to memory-like regions that precede the WMB are ordered before writes to memory-like regions that follow the WMB. Similarly, WMB guarantees that writes to non-memory-like regions that precede the WMB are ordered before writes to non-memory-like regions that follow the WMB. It does not order writes to memory-like regions relative to writes to non-memory-like regions.

WMB causes writes that are contained in buffers to be completed without unnecessary delay. It is particularly suited for batching writes to high-performance I/O devices.

WMB prevents writes that precede the WMB from being merged with writes that follow the WMB. In particular, two writes that access the same location and are separated by a WMB cause two distinct and ordered write events.

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

The WMB instruction is the preferred method for providing high-bandwidth write streams where order must be preserved between writes in that stream.

Notes:

WMB is useful for ordering streams of writes to a non-memory-like region, such as to memory-mapped control registers or to a graphics frame buffer. While both MB and WMB can ensure that writes to a non-memory-like region occur in order, without being aggregated or reordered, the WMB is usually faster and is never slower than MB.

WMB can correctly order streams of writes in programs that operate on shared sections of data if the data in those sections are protected by a classic semaphore protocol. The following example illustrates such a protocol:

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

The example above is similar to that in Section 5.5.4, except a WMB is substituted for the second MB in the lock-update-release sequence. It is correct to substitute WMB for the second MB only if:

1. All data locations that are read or written in the critical section are accessed only after acquiring a software lock by using lock_variable (and before releasing the software lock).
2. For each read *u* of shared data in the critical section, there is a write *v* such that:
   a. *v* is BEFORE the WMB
   b. *v* follows *u* in processor issue sequence (see Section 5.6.1.1)
   c. *v* either depends on *u* (see Section 5.6.1.7) or overlaps *u* (see Section 5.6.1), or both.
3. Both lock_variable and all the shared data are in memory-like regions (or lock_variable and all the shared data are in non-memory-like regions). If the lock_variable is in a non-memory-like region, the atomic lock protocol must use some implementation-specific hardware support.

The substitution of a WMB for the second MB is usually faster and never slower.

4.12 VAX Compatibility Instructions

Alpha provides the instructions shown in Table 4–18 for use in translated VAX code. These instructions are not a permanent part of the architecture and will not be available in some future implementations. They are intended to preserve customer assumptions about VAX instruction atomicity in porting code from VAX to Alpha.

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

Table 4–18: VAX Compatibility Instructions Summary

Mnemonic	Operation
RC	Read and Clear
RS	Read and Set

4.12.1 VAX Compatibility Instructions

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

RC	Read and Clear
RS	Read and Set

Qualifiers:

None

Description:

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

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

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

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

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

Notes:

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

4.13 Multimedia (Graphics and Video) Support

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

Mnemonic	Operation
MINUB8	Vector Unsigned Byte Minimum
MINSB8	Vector Signed Byte Minimum
MINUW4	Vector Unsigned Word Minimum
MINSW4	Vector Signed Word Minimum
MAXUB8	Vector Unsigned Byte Maximum
MAXSB8	Vector Signed Byte Maximum
MAXUW4	Vector Unsigned Word Maximum
MAXSW4	Vector Signed Word Maximum
PERR	Pixel Error
PKLB	Pack Longwords to Bytes
PKWB	Pack Words to Bytes
UNPKBL	Unpack Bytes to Longwords
UNPKBW	Unpack Bytes to Words

The MIN and MAX instructions allow the clamping of pixel values to maximium values that are allowed in different standards and stages of the CODECs.

The PERR instruction accelerates the macroblock search in motion estimation.

The pack and unpack (PKxB and UNPKBx) instructions accelerate the blocking of interleaved YUV coordinates for processing by the CODEC.

Implementation Note:

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

4.13.1 Byte and Word Minimum and Maximum

Format:

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

Operation:

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

Exceptions:

None

Instruction mnemonics:

MINUB8	Vector Unsigned Byte Minimum
MINSB8	Vector Signed Byte Minimum
MINUW4	Vector Unsigned Word Minimum
MINSW4	Vector Signed Word Minimum
MAXUB8	Vector Unsigned Byte Maximum
MAXSB8	Vector Signed Byte Maximum
MAXUW4	Vector Unsigned Word Maximum
MAXSW4	Vector Signed Word Maximum

Qualifiers:

None

Description:

For MINxB8, each byte of Rc is written with the smaller of the corresponding bytes of Ra or Rb. The bytes may be interpreted as signed or unsigned values.

For MINxW4, each word of Rc is written with the smaller of the corresponding words of Ra or Rb. The words may be interpreted as signed or unsigned values.

For MAXxB8, each byte of Rc is written with the larger of the corresponding bytes of Ra or Rb. The bytes may be interpreted as signed or unsigned values.

For MAXxW4, each word of Rc is written with the larger of the corresponding words of Ra or Rb. The words may be interpreted as signed or unsigned values.

4.13.2 Pixel Error

Format:

```
PERR            Ra.rq,Rb.rq,Rc.wq            ! Operate Format
```

Operation:

```
temp = 0
FOR i FROM 0 TO 7
 IF { Rav<i*8+7:i*8> GEU Rbv<i*8+7:i*8>} THEN
  temp ← temp + (Rav<i*8+7:i*8> - Rbv<i*8+7:i*8>)
 ELSE
  temp ← temp + (Rbv<i*8+7:i*8> - Rav<i*8+7:i*8>)
END
Rc ← temp
```

Exceptions:

None

Instruction mnemonics:

PERR	Pixel Error

Qualifiers:

None

Description:

The absolute value of the difference between each of the bytes in Ra and Rb is calculated. The sum of the resulting bytes is written to Rc.

4.13.3 Pack Bytes

Format:

```
PKxB            Rb.rq,Rc.wq                 ! Operate Format
```

Operation:

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

Exceptions:

None

Instruction mnemonics:

PKLB	Pack Longwords to Bytes
PKWB	Pack Words to Bytes

Qualifiers:

None

Description:

For PKLB, the component longwords of Rb are truncated to bytes and written to the lower two byte positions of Rc. The upper six bytes of Rc are written with zero.

For PKWB, the component words of Rb are truncated to bytes and written to the lower four byte positions of Rc. The upper four bytes of Rc are written with zero.

4.13.4 Unpack Bytes

Format:

UNPKBx	Rb.rq,Rc.wq	! Operate Format

Operation:

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

Exceptions:

None

Instruction mnemonics:

UNPKBL	Unpack Bytes to Longwords
UNPKBW	Unpack Bytes to Words

Qualifiers:

None

Description:

For UNPKBL, the lower two component bytes of Rb are zero-extended to longwords. The resulting longwords are written to Rc.

For UNPKBW, the lower four component bytes of Rb are zero-extended to words. The resulting words are written to Rc.