9 VAX Instruction Set

The following sections describe the native-mode instruction set. The instructions are divided into groups according to their function and are listed alphabetically within each group.

9.1 Introduction to the VAX Instruction Set

This section describes the instructions generally used by all software across all implementations of the VAX architecture.

You can find a more complete description of the instruction set in the *VAX Architecture Reference Manual*. The *VAX Architecture Reference Manual* also contains information on instructions that are generally used by privileged software and are specific to specialized portions of the VAX architecture, such as memory management, interrupts and exceptions, process dispatching, and processor registers.

A list of instructions and opcode assignments appears in Appendix D.

9.2 Instruction Descriptions

The instruction set is divided into the following 12 major sections:

- Integer arithmetic and logical
- Address
- Variable-length bit field
- Control
- Procedure call
- Miscellaneous
- Queue
- Floating point
- Character string
- Cyclic redundancy check (CRC)
- Decimal string
- Edit

Within each major section, instructions that are closely related 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, including the name and type of each instruction operand specifier and the order in which it appears in memory. Operand specifiers from left to right appear in increasing memory addresses.
- The effect on condition codes.
- Exceptions specific to the instruction. Exceptions that are generally possible for all instructions (for example, illegal or reserved addressing mode, T-bit, and memory management violations) are not listed.
- The opcodes, mnemonics, and names of each instruction in the group. The opcodes are given in hexadecimal.
- A description, in English, of the instruction.
- Optional notes on the instruction and programming examples.

Operand Specifier Notation

Operand specifiers are described as follows:

name . access-type data-type

name

A mnemonic name for the operand in the context of the instruction. The name is often abbreviated.

access-type

A letter denoting the operand specifier access type:

a	Calculate the effective address of the specified operand. Address is returned in a longword that is the actual instruction operand. Context of address calculation is given by **data-type**; that is, size to be used in autoincrement, autodecrement, and indexing.
b	No operand reference. Operand specifier is a branch displacement. Size of branch displacement is given by **data-type**.
m	Operand is read, potentially modified, and written. Note that this is *not* an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessibility.
r	Operand is read only.
v	Calculate the effective address of the specified operand. If the effective address is in memory, the address is returned in a longword that is the actual instruction operand. Context of address calculation is given by **data-type**. If the effective address is Rn, the operand is in Rn or R[n+1]'Rn.
w	Operand is written only.

data-type

A letter denoting the data type of the operand:

b	Byte
d	D_floating
f	F_floating
g	G_floating

h	H_floating
l	Longword
o	Octaword
q	Quadword
w	Word
x	First data type specified by instruction
y	Second data type specified by instruction

Operation Description Notation

The operation of an instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax. No attempt is made to formally define the syntax; it is assumed to be familiar to the reader. The notation used is an extension of the notation introduced in Section 8.6.

+	Addition
–	Subtraction, unary minus
*	Multiplication
/	Division (quotient only)
**	Exponentiation
'	Concatenation
<-	Is replaced by
=	Is defined as
Rn or R[n]	Contents of register Rn
PC, SP, FP, or AP	The contents of register R15, R14, R13, or R12, respectively
PSW	The contents of the processor status word
PSL	The contents of the processor status longword
(x)	Contents of memory location whose address is x
(x)+	Contents of memory location whose address is x; x incremented by the size of operand referenced at x
–(x)	x decremented by size of operand to be referenced at x; contents of memory location whose address is x
<x:y>	A modifier that delimits an extent from bit position x to bit position y inclusive
<x1,x2,...,xn>	A modifier that enumerates bits x1,x2,...,xn
{ }	Arithmetic parentheses used to indicate precedence
AND	Logical AND
OR	Logical OR
XOR	Logical XOR
NOT	Logical (one's) complement
LSS	Less than signed
LSSU	Less than unsigned

LEQ	Less than or equal signed
LEQU	Less than or equal unsigned
EQL	Equal signed
EQLU	Equal unsigned
NEQ	Not equal signed
NEQU	Not equal unsigned
GEQ	Greater than or equal signed
GEQU	Greater than or equal unsigned
GTR	Greater than signed
GTRU	Greater than unsigned
SEXT(x)	x is sign extended to size of operand needed
ZEXT(x)	x is zero extended to size of operand needed
REM(x,y)	Remainder of x divided by y, such that x/y and REM(x,y) have the same sign
MINU(x,y)	Minimum unsigned of x and y
MAXU(x,y)	Maximum unsigned of x and y

Use the following conventions:

- Other than alterations caused by (x)+, or –(x), and the advancement of the program counter (PC), only operands or portions of operands appearing on the left side of assignment statements are affected.
- No operator precedence is assumed, except that replacement (<-) has the lowest precedence. Precedence is indicated explicitly by { }.
- All arithmetic, logical, and relational operators are defined in the context of their operands. For example, "+" applied to floating operands means a floating add, while "+" applied to byte operands is an integer byte add. Similarly, "LSS" is a floating comparison when applied to floating operands, while "LSS" is an integer byte comparison when applied to byte operands.
- Instruction operands are evaluated according to the operand specifier conventions (see Chapter 8). The order in which operands appear in the instruction description has no effect on the order of evaluation.
- Condition codes generally indicate the effect of an operation on the value of actual stored results, not on "true" results (which might be generated internally to greater precision). For example, two positive integers can be added together and the sum stored as a negative value because of overflow. The condition codes indicate a negative value even though the "true" result is clearly positive.

9.2.1 Integer Arithmetic and Logical Instructions

The following instructions are described in this section:

	Description and Opcode	Number of Instructions
1.	Add Aligned Word ADAWI add.rw, sum.mw	1
2.	Add 2 Operand ADD{B,W,L}2 add.rx, sum.mx	3
3.	Add 3 Operand ADD{B,W,L}3 add1.rx, add2.rx, sum.wx	3
4.	Add with Carry ADWC add.rl, sum.ml	1
5.	Arithmetic Shift ASH{L,Q} cnt.rb, src.rx, dst.wx	2
6.	Bit Clear 2 Operand BIC{B,W,L}2 mask.rx, dst.mx	3
7.	Bit Clear 3 Operand BIC{B,W,L}3 mask.rx, src.rx, dst.wx	3
8.	Bit Set 2 Operand BIS{B,W,L}2 mask.rx, dst.mx	3
9.	Bit Set 3 Operand BIS{B,W,L}3 mask.rx, src.rx, dst.wx	3
10.	Bit Test BIT{B,W,L} mask.rx, src.rx	3
11.	Clear CLR{B,W,L,Q,O} dst.wx	5
12.	Compare CMP{B,W,L} src1.rx, src2.rx	3
13.	Convert CVT{B,W,L}{B,W,L} src.rx, dst.wy All pairs except BB,WW,LL	6
14.	Decrement DEC{B,W,L} dif.mx	3
15.	Divide 2 Operand DIV{B,W,L}2 divr.rx, quo.mx	3
16.	Divide 3 Operand DIV{B,W,L}3 divr.rx, divd.rx, quo.wx	3
17.	Extended Divide EDIV divr.rl, divd.rq, quo.wl, rem.wl	1
18.	Extended Multiply EMUL mulr.rl, muld.rl, add.rl, prod.wq	1
19.	Increment INC{B,W,L} sum.mx	3
20.	Move Complemented MCOM{B,W,L} src.rx, dst.wx	3

	Description and Opcode	Number of Instructions
21.	Move Negated MNEG{B,W,L} src.rx, dst.wx	3
22.	Move OV{B,W,L,Q} src.rx, dst.wx	4
23.	Move Zero-Extended MOVZ{BW,BL,WL} src.rx, dst.wy	3
24.	Multiply 2 Operand MUL{B,W,L}2 mulr.rx, prod.mx	3
25.	Multiply 3 Operand MUL{B,W,L}3 mulr.rx, muld.rx, prod.wx	3
26.	Push Long PUSHL src.rl, {-(SP).wl}	1
27.	Rotate Long ROTL cnt.rb, src.rl, dst.wl	1
28.	Subtract with Carry SBWC sub.rl, dif.ml	1
29.	Subtract 2 Operand SUB{B,W,L}2 sub.rx, dif.mx	3
30.	Subtract 3 Operand SUB{B,W,L}3 sub.rx, min.rx, dif.wx	3
31.	Test TST{B,W,L} src.rx	3
32.	Exclusive OR 2 Operand XOR{B,W,L}2 mask.rx, dst.mx	3
33.	Exclusive OR 3 Operand XOR{B,W,L}3 mask.rx, src.rx, dst.wx	3

ADAWI

Add Aligned Word Interlocked

FORMAT

opcode add.rw, sum.mw

condition codes

N	⟵ sum LSS 0;
Z	⟵ sum EQL 0;
V	⟵ {integer overflow};
C	⟵ {carry from most-significant bit};

exceptions

reserved operand fault
integer overflow

opcodes

58	ADAWI	Add Aligned Word Interlocked

DESCRIPTION

The addend operand is added to the sum operand, and the sum operand is replaced by the result. The operation is interlocked against similar operations on other processors in a multiprocessor system. The destination must be aligned on a word boundary; that is, bit 0 of the address of the sum operand must be zero. If it is not, a reserved operand fault is taken.

Notes

1 Integer overflow occurs if the input operands to the add have the same sign, and the result has the opposite sign. On overflow, the sum operand is replaced by the low-order bits of the true result.

2 If the addend and the sum operands overlap, the result and the condition codes are UNPREDICTABLE.

ADD

Add

FORMAT

2operand:	*opcode*	*add.rx, sum.mx*
3operand:	*opcode*	*add1.rx, add2.rx, sum.wx*

condition codes

N	⟵ sum LSS 0;
Z	⟵ sum EQL 0;
V	⟵ {integer overflow};
C	⟵ {carry from most-significant bit};

exceptions

integer overflow

opcodes

80	ADDB2	Add Byte 2 Operand
81	ADDB3	Add Byte 3 Operand
A0	ADDW2	Add Word 2 Operand
A1	ADDW3	Add Word 3 Operand
C0	ADDL2	Add Long 2 Operand
C1	ADDL3	Add Long 3 Operand

DESCRIPTION

In 2 operand format, the addend operand is added to the sum operand and the sum operand is replaced by the result. In 3 operand format, the addend 1 operand is added to the addend 2 operand and the sum operand is replaced by the result.

Note

Integer overflow occurs if the input operands to the add have the same sign and the result has the opposite sign. On overflow, the sum operand is replaced by the low-order bits of the true result.

ADWC

Add with Carry

FORMAT

opcode *add.rl, sum.ml*

condition codes

N	⟵ sum LSS 0;
Z	⟵ sum EQL 0;
V	⟵ {integer overflow};
C	⟵ {carry from most-significant bit};

exceptions

integer overflow

opcodes

D8	ADWC	Add with Carry

DESCRIPTION

The contents of the condition code C-bit and the addend operand are added to the sum operand and the sum operand is replaced by the result.

Notes

1. On overflow, the sum operand is replaced by the low-order bits of the true result.
2. The two additions in the operation are performed simultaneously.

ASH

Arithmetic Shift

FORMAT

opcode *cnt.rb, src.rx, dst.wx*

condition codes

N	⟵ dst LSS 0;
Z	⟵ dst EQL 0;
V	⟵ {integer overflow};
C	⟵ 0;

exceptions

integer overflow

opcodes

78	ASHL	Arithmetic Shift Long
79	ASHQ	Arithmetic Shift Quad

DESCRIPTION

The source operand is arithmetically shifted by the number of bits specified by the count operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand shifts to the left, bringing zeros into the least significant bit. A negative count operand shifts to the right, bringing in copies of the most significant (sign) bit into the most significant bit. A zero count operand replaces the destination operand with the unshifted source operand.

Notes

1 Integer overflow occurs on a left shift if any bit shifted into the sign bit position differs from the sign bit of the source operand.

2 If **cnt** GTR 32 (ASHL) or **cnt** GTR 64 (ASHQ), the destination operand is replaced by zero.

3 If **cnt** LEQ –31 (ASHL) or **cnt** LEQ –63 (ASHQ), all the bits of the destination operand are copies of the sign bit of the source operand.

BIC

Bit Clear

FORMAT

2operand: *opcode* *mask.rx, dst.mx*
3operand: *opcode* *mask.rx, src.rx, dst.wx*

condition codes

N ⟵ dst LSS 0;
Z ⟵ dst EQL 0;
V ⟵ 0;
C ⟵ C;

exceptions

None.

opcodes

8A	BICB2	Bit Clear Byte
8B	BICB3	Bit Clear Byte
AA	BICW2	Bit Clear Word
AB	BICW3	Bit Clear Word
CA	BICL2	Bit Clear Long
CB	BICL3	Bit Clear Long

DESCRIPTION

In 2 operand format, the result of the logical AND on the destination operand and the one's complement of the mask operand replaces the destination operand. In 3 operand format, the result of the logical AND on the source operand and the one's complement of the mask operand replaces the destination operand.

BIS

Bit Set

FORMAT

2operand:	*opcode*	*mask.rx, dst.mx*
3operand:	*opcode*	*mask.rx, src.rx, dst.wx*

condition codes

N	$\longleftarrow$ dst LSS 0;
Z	$\longleftarrow$ dst EQL 0;
V	$\longleftarrow$ 0;
C	$\longleftarrow$ C;

exceptions

None.

opcodes

88	BISB2	Bit Set Byte 2 Operand
89	BISB3	Bit Set Byte 3 Operand
A8	BISW2	Bit Set Word 2 Operand
A9	BISW3	Bit Set Word 3 Operand
C8	BISL2	Bit Set Long 2 Operand
C9	BISL3	Bit Set Long 3 Operand

DESCRIPTION

In 2 operand format, the result of the logical OR on the mask operand and the destination operand replaces the destination operand. In 3 operand format, the result of the logical OR on the mask operand and the source operand replaces the destination operand.

BIT

Bit Test

FORMAT

opcode mask.rx, src.rx

condition codes

N	⟵ tmp LSS 0;
Z	⟵ tmp EQL 0;
V	⟵ 0;
C	⟵ C;

exceptions

None.

opcodes

93	BITB	Bit Test Byte
B3	BITW	Bit Test Word
D3	BITL	Bit Test Long

DESCRIPTION

The logical AND is performed on the mask operand and the source operand. Both operands are unaffected. The only action is to modify condition codes.

CLR

Clear

FORMAT

opcode *dst.wx*

condition codes

N ⟵ 0;
Z ⟵ 1;
V ⟵ 0;
C ⟵ C;

exceptions

None.

opcodes

94	CLRB	Clear Byte
B4	CLRW	Clear Word
D4	CLRL	Clear Long
7C	CLRQ	Clear Quad
7CFD	CLRO	Clear Octa

DESCRIPTION

The destination operand is replaced by zero.

Note

CLR*x* **dst** is equivalent to MOV*x* S^#0, **dst**, but is 1 byte shorter.

CMP

Compare

FORMAT

opcode *src1.rx, src2.rx*

condition codes

N	⟵ src1 LSS src2;
Z	⟵ src1 EQL src2;
V	⟵ 0;
C	⟵ src1 LSSU src2;

exceptions

None.

opcodes

91	CMPB	Compare Byte
B1	CMPW	Compare Word
D1	CMPL	Compare Long

DESCRIPTION

The source 1 operand is compared with the source 2 operand. The only action is to modify the condition codes.

CVT

Convert

FORMAT

opcode src.rx, dst.wy

condition codes

N	⟵ dst LSS 0;
Z	⟵ dst EQL 0;
V	⟵ {integer overflow};
C	⟵ 0;

exceptions

integer overflow

opcodes

99	CVTBW	Convert Byte to Word
98	CVTBL	Convert Byte to Long
33	CVTWB	Convert Word to Byte
32	CVTWL	Convert Word to Long
F6	CVTLB	Convert Long to Byte
F7	CVTLW	Convert Long to Word

DESCRIPTION

The source operand is converted to the data type of the destination operand and the destination operand is replaced by the result. Conversion of a shorter data type to a longer one is done by sign extension; conversion of longer data type to a shorter one is done by truncation of the higher-numbered (most significant) bits.

Note

Integer overflow occurs if any truncated bits of the source operand are not equal to the sign bit of the destination operand.

DEC

Decrement

FORMAT

opcode *dif.mx*

condition codes

N	⟵ dif LSS 0;
Z	⟵ dif EQL 0;
V	⟵ {integer overflow};
C	⟵ {borrow into most significant bit};

exceptions

integer overflow

opcodes

97	DECB	Decrement Byte
B7	DECW	Decrement Word
D7	DECL	Decrement Long

DESCRIPTION

One is subtracted from the difference operand, and the difference operand is replaced by the result.

Notes

1 Integer overflow occurs if the largest negative integer is decremented. On overflow, the difference operand is replaced by the largest positive integer.

2 DEC*x* **dif** is equivalent to SUB*x* S^#1, **dif**, but is 1 byte shorter.

DIV

Divide

FORMAT

2operand: opcode divr.rx, quo.mx
3operand: opcode divr.rx, divd.rx, quo.wx

condition codes

N	⟵ quo LSS 0;
Z	⟵ quo EQL 0;
V	⟵ {integer overflow} OR {divr EQL 0};
C	⟵ 0;

exceptions

integer overflow
divide by zero

opcodes

86	DIVB2	Divide Byte 2 Operand
87	DIVB3	Divide Byte 3 Operand
A6	DIVW2	Divide Word 2 Operand
A7	DIVW3	Divide Word 3 Operand
C6	DIVL2	Divide Long 2 Operand
C7	DIVL3	Divide Long 3 Operand

DESCRIPTION

In 2 operand format, the quotient operand is divided by the divisor operand, and the quotient operand is replaced by the result. In 3 operand format, the dividend operand is divided by the divisor operand, and the quotient operand is replaced by the result.

Notes

1. Division is performed so that the remainder has the same sign as the dividend; that is, the result is truncated toward zero. (Note that a remainder of zero is not saved.)
2. Integer overflow occurs only if the largest negative integer is divided by –1. On overflow, operands are affected as in note 3 following.
3. If the divisor operand is zero, then in 2 operand format the quotient operand is not affected; in 3 operand format the quotient operand is replaced by the dividend operand.

EDIV

Extended Divide

FORMAT

opcode divr.rl, divd.rq, quo.wl, rem.wl

condition codes

N	← quo LSS 0;
Z	← quo EQL 0;
V	← {integer overflow} OR {divr EQL 0};
C	← 0;

exceptions

integer overflow
divide by zero

opcodes

7B	EDIV	Extended Divide

DESCRIPTION

The dividend operand is divided by the divisor operand, the quotient operand is replaced by the quotient, and the remainder operand is replaced by the remainder.

Notes

1 The division is performed such that the remainder operand (unless it is zero) has the same sign as the dividend operand.

2 On overflow, the operands are affected as in note 3, following.

3 If the divisor operand is zero, then the quotient operand is replaced by bits 31:0 of the dividend operand, and the remainder operand is replaced by zero.

EMUL

Extended Multiply

FORMAT

opcode *mulr.rl, muld.rl, add.rl, prod.wq*

condition codes

N	⟵ prod LSS 0;
Z	⟵ prod EQL 0;
V	⟵ 0;
C	⟵ 0;

exceptions

None.

opcodes

7A	EMUL	Extended Multiply

DESCRIPTION

The multiplicand operand is multiplied by the multiplier operand, giving a double-length result. The addend operand is sign extended to double length and added to the result. The product operand is replaced by the final result.

INC

Increment

FORMAT

opcode sum.mx

condition codes

N	⟵ sum LSS 0;
Z	⟵ sum EQL 0;
V	⟵ {integer overflow};
C	⟵ {carry from most significant bit};

exceptions

integer overflow

opcodes

96	INCB	Increment Byte
B6	INCW	Increment Word
D6	INCL	Increment Long

DESCRIPTION

One is added to the sum operand and the sum operand is replaced by the result.

Notes

1 Arithmetic overflow occurs if the largest positive integer is incremented. On overflow, the sum operand is replaced by the largest negative integer.

2 INC*x* **sum** is equivalent to ADD*x* S^#1, **sum**, but is 1 byte shorter.

MCOM

Move Complemented

FORMAT

opcode   *src.rx, dst.wx*

condition codes

N	⟵ dst LSS 0;
Z	⟵ dst EQL 0;
V	⟵ 0;
C	⟵ C;

exceptions

None.

opcodes

92	MCOMB	Move Complemented Byte
B2	MCOMW	Move Complemented Word
D2	MCOML	Move Complemented Long

DESCRIPTION

The destination operand is replaced by the one's complement of the source operand.

MNEG

Move Negated

FORMAT

opcode *src.rx, dst.wx*

condition codes

N	⟵ dst LSS 0;
Z	⟵ dst EQL 0;
V	⟵ {integer overflow};
C	⟵ dst NEQ 0;

exceptions

integer overflow

opcodes

8E	MNEGB	Move Negated Byte
AE	MNEGW	Move Negated Word
CE	MNEGL	Move Negated Long

DESCRIPTION

The destination operand is replaced by the negative of the source operand.

Note

Integer overflow occurs if the source operand is the largest negative integer (which has no positive counterpart). On overflow, the destination operand is replaced by the source operand.

MOV

Move

FORMAT *opcode src.rx, dst.wx*

condition codes

```
N   ←— dst LSS 0;
Z   ←— dst EQL 0;
V   ←— 0;
C   ←— C;
```

exceptions

None.

opcodes

90	MOVB	Move Byte
B0	MOVW	Move Word
D0	MOVL	Move Long
7D	MOVQ	Move Quad
7DFD	MOVO	Move Octa

DESCRIPTION The destination operand is replaced by the source operand.

MOVZ

Move Zero-Extended

FORMAT

opcode src.rx, dst.wy

condition codes

N	⟵ 0;
Z	⟵ dst EQL 0;
V	⟵ 0;
C	⟵ C;

exceptions

None.

opcodes

9B	MOVZBW	Move Zero-Extended Byte to Word
9A	MOVZBL	Move Zero-Extended Byte to Long
3C	MOVZWL	Move Zero-Extended Word to Long

DESCRIPTION

For MOVZBW, bits 7:0 of the destination operand are replaced by the source operand; bits 15:8 are replaced by zero. For MOVZBL, bits 7:0 of the destination operand are replaced by the source operand; bits 31:8 are replaced by zero. For MOVZWL, bits 15:0 of the destination operand are replaced by the source operand; bits 31:16 are replaced by zero.

MUL

Multiply

FORMAT

2operand: opcode mulr.rx, prod.mx
3operand: opcode mulr.rx, muld.rx, prod.wx

condition codes

N	⟵ prod LSS 0;
Z	⟵ prod EQL 0;
V	⟵ {integer overflow};
C	⟵ 0;

exceptions

integer overflow

opcodes

84	MULB2	Multiply Byte 2 Operand
85	MULB3	Multiply Byte 3 Operand
A4	MULW2	Multiply Word 2 Operand
A5	MULW3	Multiply Word 3 Operand
C4	MULL2	Multiply Long 2 Operand
C5	MULL3	Multiply Long 3 Operand

DESCRIPTION

In 2 operand format, the product operand is multiplied by the multiplier operand, and the product operand is replaced by the low half of the double-length result. In 3 operand format, the multiplicand operand is multiplied by the multiplier operand, and the product operand is replaced by the low half of the double-length result.

Note

Integer overflow occurs if the high half of the double-length result is not equal to the sign extension of the low half of the double-length result.

PUSHL

Push Long

FORMAT

opcode *src.rl*

condition codes

N	⟵ src LSS 0;
Z	⟵ src EQL 0;
V	⟵ 0;
C	⟵ C;

exceptions

None.

opcodes

DD	PUSHL	Push Long

DESCRIPTION

The longword source operand is pushed on the stack.

Notes

1. PUSHL is equivalent to MOVL **src**, –(SP), but is 1 byte shorter.
2. POPL is not a VAX instruction. However, the assembler recognizes the inclusion of *POPL destination* in a program, for which it generates the code for *MOVL (SP)+,destination.*

ROTL

Rotate Long

FORMAT

opcode cnt.rb, src.rl, dst.wl

condition codes

N	← dst LSS 0;
Z	← dst EQL 0;
V	← 0;
C	← C;

exceptions

None.

opcodes

9C	ROTL	Rotate Long

DESCRIPTION

The source operand is rotated logically by the number of bits specified by the count operand, and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand rotates to the left. A negative count operand rotates to the right. A zero count operand replaces the destination operand with the source operand.

SBWC

Subtract with Carry

FORMAT

opcode sub.rl, dif.ml

condition codes

N	⟵ dif LSS 0;
Z	⟵ dif EQL 0;
V	⟵ {integer overflow};
C	⟵ {borrow into most significant bit};

exceptions

integer overflow

opcodes

D9	SBWC	Subtract with carry

DESCRIPTION

The subtrahend operand and the contents of the condition code C-bit are subtracted from the difference operand, and the difference operand is replaced by the result.

Notes

1. On overflow, the difference operand is replaced by the low-order bits of the true result.
2. The two subtractions in the operation are performed simultaneously.

SUB

Subtract

FORMAT

2operand: opcode sub.rx, dif.mx
3operand: opcode sub.rx, min.rx, dif.wx

condition codes

N	⟵ dif LSS 0;
Z	⟵ dif EQL 0;
V	⟵ {integer overflow};
C	⟵ {borrow into most significant bit};

exceptions

integer overflow

opcodes

82	SUBB2	Subtract Byte 2 Operand
83	SUBB3	Subtract Byte 3 Operand
A2	SUBW2	Subtract Word 2 Operand
A3	SUBW3	Subtract Word 3 Operand
C2	SUBL2	Subtract Long 2 Operand
C3	SUBL3	Subtract Long 3 Operand

DESCRIPTION

In 2 operand format, the subtrahend operand is subtracted from the difference operand, and the difference operand is replaced by the result. In 3 operand format, the subtrahend operand is subtracted from the minuend operand, and the difference operand is replaced by the result.

Note

Integer overflow occurs if the input operands to the subtract are of different signs and the sign of the result is the sign of the subtrahend. On overflow, the difference operand is replaced by the low-order bits of the true result.

TST

Test

FORMAT

opcode *src.rx*

condition codes

```
N  ← src LSS 0;
Z  ← src EQL 0;
V  ← 0;
C  ← 0;
```

exceptions

None.

opcodes

95	TSTB	Test Byte
B5	TSTW	Test Word
D5	TSTL	Test Long

DESCRIPTION

The condition codes are modified according to the value of the source operand.

Note

The operand **src** is equivalent to CMP*x* **src**, S^#0, but is 1 byte shorter.

XOR

Exclusive OR

FORMAT

2operand:	*opcode*	*mask.rx, dst.mx*
3operand:	*opcode*	*mask.rx, src.rx, dst.wx*

condition codes

N	⟵ dst LSS 0;
Z	⟵ dst EQL 0;
V	⟵ 0;
C	⟵ C;

exceptions

None.

opcodes

8C	XORB2	Exclusive OR Byte 2 Operand
8D	XORB3	Exclusive OR Byte 3 Operand
AC	XORW2	Exclusive OR Word 2 Operand
AD	XORW3	Exclusive OR Word 3 Operand
CC	XORL2	Exclusive OR Long 2 Operand
CD	XORL3	Exclusive OR Long 3 Operand

DESCRIPTION

In 2 operand format, the result of the logical XOR on the mask operand and the destination operand replaces the destination operand. In 3 operand format, the result of the logical XOR on the mask operand and the source operand replaces the destination operand.

9.2.2 Address Instructions

The following instructions are described in this section.

	Description and Opcode	Number of Instructions
1.	Move Address MOVA{B,W,L=F,Q=D=G,O=H} src.ax, dst.wl	5
2.	Push Address PUSHA{B,W,L=F,Q=D=G,O=H} src.ax, {-(SP).wl}	5

MOVA

Move Address

FORMAT

opcode src.ax, dst.wl

condition codes

N	⟵ dst LSS 0;
Z	⟵ dst EQL 0;
V	⟵ 0;
C	⟵ C;

exceptions

None.

opcodes

9E	MOVAB	Move Address Byte
3E	MOVAW	Move Address Word
DE	MOVAL	Move Address Long
	MOVAF	Move Address F_floating
7E	MOVAQ	Move Address Quad
	MOVAD	Move Address D_floating
	MOVAG	Move Address G_floating
7EFD	MOVAH	Move Address H_floating
	MOVAO	Move Address Octa

DESCRIPTION

The destination operand is replaced by the source operand. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address replaces the destination operand is not referenced.

Note

The access type of the source operand is address, which causes the address of the specified operand to be moved.

PUSHA

Push Address

FORMAT

opcode *src.ax*

condition codes

N	⟵ src LSS 0;
Z	⟵ src EQL 0;
V	⟵ 0;
C	⟵ C;

exceptions

None.

opcodes

9F	PUSHAB	Push Address Byte
3F	PUSHAW	Push Address Word
DF	PUSHAL	Push Address Long,
	PUSHAF	Push Address F_floating
7F	PUSHAQ	Push Address Quad,
	PUSHAD	Push Address D_floating,
	PUSHAG	Push Address G_floating
7FFD	PUSHAH	Push Address H_floating
	PUSHAO	Push Address Octa

DESCRIPTION

The source operand is pushed on the stack. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address is pushed is not referenced.

Notes

1 PUSHA*x* **src** is equivalent to MOVA*x* **src**, –(SP), but is one byte shorter.

2 The source operand is of address access type, which causes the address of the specified operand to be pushed.

9.2.3 Variable-Length Bit Field Instructions

A variable-length bit field is specified by the following three operands:

1 A longword position operand.

2 A byte field size operand in the range 0 to 32; if out of this range, a reserved operand fault occurs.

3 A base address. Use the position operand to locate the bit field relative to this base address. The address is obtained from an operand of address access type. However, unlike other instances of operand specifiers of address access type, register mode can be designated in the operand specifier. In this case, the field is contained in the register n designated by the operand specifier (or register n+1 concatenated with register n). (See Chapter 8.) If the field is contained in a register and the size operand is not zero, the position operand must have a value in the range 0 to 31, or a reserved operand fault occurs.

Zero bytes are referenced if the field size is zero.

The following instructions are described in this section.

	Description and Opcode	Number of Instructions
1.	Compare Field CMPV pos.rl, size.rb, base.vb, {field.rv}, src.rl	1
2.	Compare Zero-Extended Field CMPZV pos.rl, size.rb, base.vb, {field.rv}, src.rl	1
3.	Extract Field EXTV pos.rl, size.rb, base.vb, {field.rv}, dst.wl	1
4.	Extract Zero-Extended Field EXTZV pos.rl, size.rb, base.vb, {field.rv}, dst.wl	1
5.	Find First FF{S,C} startpos.rl, size.rb, base.vb, {field.rv}, findpos.wl	2
6.	Insert Field INSV src.rl, pos.rl, size.rb, base.vb, {field.wv}	1

The following variable-length bit field instructions are described in Section 9.2.4:

	Description and Opcode	Number of Instructions
1.	Branch on Bit BB{S,C} pos.rl, base.vb, displ.bb, {field.rv}	2
2.	Branch on Bit (and modify without interlock) BB{S,C}{S,C} pos.rl, base.vb, displ.bb, {field.mv}	4
3.	Branch on Bit (and modify) Interlocked BB{SS,CC}I pos.rl, base.vb, displ.bb, {field.mv}	2

CMP

Compare Field

FORMAT *opcode pos.rl, size.rb, base.vb, src.rl*

condition codes

N	⟵ tmp LSS src;
Z	⟵ tmp EQL src;
V	⟵ 0;
C	⟵ tmp LSSU src;

exceptions

reserved operand

opcodes

EC	CMPV	Compare Field
ED	CMPZV	Compare Zero-Extended Field

DESCRIPTION

The field specified by the position, size, and base operands is compared with the source operand. For CMPV, the source operand is compared with the sign-extended field. For CMPZV, the source operand is compared with the zero-extended field. The only action is to affect the condition codes.

Notes

1 A reserved operand fault occurs if:
- **size** GTRU 32
- **pos** GTRU 31, **size** NEQ 0, and the field is contained in the registers

2 On a reserved operand fault, the condition codes are UNPREDICTABLE.

EXT

Extract Field

FORMAT *opcode pos.rl, size.rb, base.vb, dst.wl*

condition codes

N	⟵ dst LSS 0;
Z	⟵ dst EQL 0;
V	⟵ 0;
C	⟵ C;

exceptions

reserved operand

opcodes

EE	EXTV	Extract Field
EF	EXTZV	Extract Zero-Extended Field

DESCRIPTION For EXTV, the destination operand is replaced by the sign-extended field specified by the position, size, and base operands. For EXTZV, the destination operand is replaced by the zero-extended field specified by the position, size, and base operands. If the size operand is zero, the only action is to replace the destination operand with zero and to modify the condition codes.

Notes

1 A reserved operand fault occurs if:
- **size** GTRU 32
- **pos** GTRU 31, **size** NEQ 0, and the field is contained in the registers

2 On a reserved operand fault, the destination operand is unaffected, and the condition codes are UNPREDICTABLE.

FF

Find First

FORMAT

opcode startpos.rl, size.rb, base.vb, findpos.wl

condition codes

N	⟵ 0;
Z	⟵ {bit not found};
V	⟵ 0;
C	⟵ 0;

exceptions

reserved operand

opcodes

EB	FFC	Find First Clear
EA	FFS	Find First Set

DESCRIPTION

A field specified by the start position, size, and base operands is extracted. Starting at bit 0 and extending to the highest bit in the field, the field is tested for a bit in the state indicated by the instruction. If a bit in the indicated state is found, the find position operand is replaced by the position of the bit, and the Z condition code bit is cleared. If no bit in the indicated state is found, the find position operand is replaced by the position (relative to the base) of a bit one position to the left of the specified field, and the Z condition code bit is set. If the size operand is zero, the find position operand is replaced by the start position operand, and the Z condition code bit is set.

Notes

1 A reserved operand fault occurs if:

- **size** GTRU 32
- **startpos** GTRU 31, **size** NEQ 0, and the field is contained in the registers

2 On a reserved operand fault, the find position operand is unaffected, and the condition codes are UNPREDICTABLE.

INSV

Insert Field

FORMAT

opcode *src.rl, pos.rl, size.rb, base.vb*

condition codes

N	← N;
Z	← Z;
V	← V;
C	← C;

exceptions

reserved operand

opcodes

F0	INSV	Insert Field

DESCRIPTION

The field specified by the position, size, and base operands is replaced by bits *size* − 1 : 0 of the source operand. If the size operand is zero, the instruction has no effect.

Notes

1 A reserved operand fault occurs if:
   - **size** GTRU 32
   - **pos** GTRU 31, **size** NEQ 0, and the field is contained in the registers

2 On a reserved operand fault, the field is unaffected, and the condition codes are UNPREDICTABLE.

9.2.4 Control Instructions

In most implementations of the VAX architecture, improved execution speed will result if the target of a control instruction is on an aligned longword boundary.

The following instructions are described in this section.

	Description and Opcode	Number of Instructions
1.	Add Compare and Branch ACB{B,W,L,F,D,G,H} limit.rx, add.rx, index.mx, displ.bw Compare is LE on positive add, GE on negative add.	7
2.	Add One and Branch Less Than or Equal AOBLEQ limit.rl, index.ml, displ.bb	1
3.	Add One and Branch Less Than AOBLSS limit.rl, index.ml, displ.bb	1
4.	Conditional Branch	12

Condition	Name
LSS	Less Than
LEQ	Less Than or Equal
EQL, EQLU	Equal, Equal Unsigned
NEQ, NEQU	Not Equal, Not Equal Unsigned
GEQ	Greater Than or Equal
GTR	Greater Than
LSSU, CS	Less Than Unsigned, Carry Set
LEQU	Less Than or Equal Unsigned
GEQU, CC	Greater Than or Equal Unsigned, Carry Clear
GTRU	Greater Than Unsigned
VS	Overflow Set
VC	Overflow Clear

	Description and Opcode	Number of Instructions
5.	Branch on Bit BB{S,C} pos.rl, base.vb, displ.bb, {field.rv}	2
6.	Branch on Bit (and modify without interlock) BB{S,C}{S,C} pos.rl, base.vb, displ.bb, {field.mv}	4
7.	Branch on Bit (and modify) Interlocked BB{SS,CC}I pos.rl, base.vb, displ.bb, {field.mv}	2

	Description and Opcode	Number of Instructions
8.	Branch on Low Bit BLB{S,C} src.rl, displ.bb	2
9.	Branch with {Byte, Word} Displacement BR{B,W} displ.bx	2
10.	Branch to Subroutine with {Byte, Word} Displacement BSB{B,W} displ.bx, {–(SP).wl}	2
11.	Case CASE{B,W,L} selector.rx, base.rx, limit.rx, displ.bw-list	3
12.	Jump JMP dst.ab	1
13.	Jump to Subroutine JSB dst.ab, {–(SP).wl}	1
14.	Return from Subroutine RSB {(SP)+.rl}	1
15.	Subtract One and Branch Greater Than or Equal SOBGEQ index.ml, displ.bb	1
16.	Subtract One and Branch Greater Than SOBGTR index.ml, displ.bb	1

ACB

Add Compare and Branch

FORMAT

opcode limit.rx, add.rx, index.mx, displ.bw

condition codes

N	←	index LSS 0;
Z	←	index EQL 0;
V	←	{integer overflow};
C	←	C;

exceptions

integer overflow
floating overflow
floating underflow
reserved operand

opcodes

9D	ACBB	Add Compare and Branch Byte
3D	ACBW	Add Compare and Branch Word
F1	ACBL	Add Compare and Branch Long
4F	ACBF	Add Compare and Branch F_floating
4FFD	ACBG	Add Compare and Branch G_floating
6F	ACBD	Add Compare and Branch D_floating
6FFD	ACBH	Add Compare and Branch H_floating

DESCRIPTION

The addend operand is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If the addend operand is positive (or zero) and the comparison is less than or equal to zero, or if the addend is negative and the comparison is greater than or equal to zero, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result.

Notes

1 ACB efficiently implements the general FOR or DO loops in high-level languages, since the sense of the comparison between **index** and **limit** is dependent on the sign of the addend.

2 On integer overflow, the index operand is replaced by the low-order bits of the true result. Comparison and branch determination proceed normally on the updated index operand.

3 On floating underflow, if FU is clear, the index operand is replaced by zero, and comparison and branch determination proceed normally. A fault occurs if FU is set, and the index operand is unaffected.

4 On floating overflow, the instruction takes a floating overflow fault, and the index operand is unaffected.

5 On a reserved operand fault, the index operand is unaffected, and condition codes are UNPREDICTABLE.

6 Except for the circumstance described in note 5, the C-bit is unaffected.

AOBLEQ

Add One and Branch Less Than or Equal

FORMAT

opcode *limit.rl, index.ml, displ.bb*

condition codes

N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C;

exceptions

integer overflow

opcodes

F3	AOBLEQ	Add One and Branch Less Than or Equal

DESCRIPTION

One is added to the index operand, and the index operand is replaced by the result. The index operand is compared with the limit operand. If the comparison is less than or equal to zero, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result.

Notes

1 Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer, and the branch is taken.

2 The C-bit is unaffected.

AOBLSS

Add One and Branch Less Than

FORMAT

opcode   limit.rl, index.ml, displ.bb

condition codes

N	⟵ index LSS 0;
Z	⟵ index EQL 0;
V	⟵ {integer overflow};
C	⟵ C;

exceptions

integer overflow

opcodes

F2	AOBLSS	Add One and Branch Less Than

DESCRIPTION

One is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If the comparison result is less than zero, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result.

Notes

1 Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer, and thus (unless the limit operand is the largest negative integer), the branch is taken.

2 The C-bit is unaffected.

B

Branch on (condition)

FORMAT *opcode displ.bb*

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

None.

opcodes

14	{N OR Z} EQL 0	BGTR	Branch on Greater Than (signed)
15	{N OR Z} EQL 1	BLEQ	Branch on Less Than or Equal (signed)
12	Z EQL 0	BNEQ,	Branch on Not Equal (signed)
		BNEQU	Branch on Not Equal Unsigned
13	Z EQL 1	BEQL,	Branch on Equal (signed)
		BEQLU	Branch on Equal Unsigned
18	N EQL 0	BGEQ	Branch on Greater Than or Equal (signed)
19	N EQL 1	BLSS	Branch on Less Than (signed)
1A	{C OR Z} EQL 0	BGTRU	Branch on Greater Than Unsigned
1B	{C OR Z} EQL 1	BLEQU	Branch Less Than or Equal Unsigned
1C	V EQL 0	BVC	Branch on Overflow Clear
1D	V EQL 1	BVS	Branch on Overflow Set
1E	C EQL 0	BGEQU,	Branch on Greater Than or Equal Unsigned
		BCC	Branch on Carry Clear
1F	C EQL 1	BLSSU,	Branch on Less Than Unsigned
		BCS	Branch on Carry Set

DESCRIPTION The condition codes are tested. If the condition indicated by the instruction is met, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result.

Notes

The VAX conditional branch instructions permit considerable flexibility in branching but require care in choosing the correct branch instruction. The conditional branch instructions are best seen as three overlapping groups:

1 Overflow and Carry Group

BVS	V EQL 1
BVC	V EQL 0
BCS	C EQL 1
BCC	C EQL 0

Typically, you would use these instructions to check for overflow (when overflow traps are not enabled), for multiprecision arithmetic, and for other special purposes.

2 Unsigned Group

BLSSU	C EQL 1
BLEQU	{C OR Z} EQL 1
BEQLU	Z EQL 1
BNEQU	Z EQL 0
BGEQU	C EQL 0
BGTRU	{C OR Z} EQL 0

These instructions typically follow integer and field instructions where the operands are treated as unsigned integers, address instructions, and character string instructions.

3 Signed Group

BLSS	N EQL 1
BLEQ	{N OR Z} EQL 1
BEQL	Z EQL 1
BNEQ	Z EQL 0
BGEQ	N EQL 0
BGTR	{N OR Z} EQL 0

These instructions typically follow floating-point instructions, decimal string instructions, and integer and field instructions where the operands are being treated as signed integers.

BB

Branch on Bit

FORMAT

opcode pos.rl, base.vb, displ.bb

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

reserved operand

opcodes

E0	BBS	Branch on Bit Set
E1	BBC	Branch on Bit Clear

DESCRIPTION

The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result.

Notes

1 A reserved operand fault occurs if **pos** GTRU 31 and the bit specified is contained in a register.

2 On a reserved operand fault, the condition codes are UNPREDICTABLE.

BB

Branch on Bit (and modify without interlock)

FORMAT

opcode pos.rl, base.vb, displ.bb

condition codes

N	⟵ N;
Z	⟵ Z;
V	⟵ V;
C	⟵ C;

exceptions

reserved operand

opcodes

E2	BBSS	Branch on Bit Set and Set
E3	BBCS	Branch on Bit Clear and Set
E4	BBSC	Branch on Bit Set and Clear
E5	BBCC	Branch on Bit Clear and Clear

DESCRIPTION

The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result. Regardless of whether the branch is taken or not, the tested bit is put in the new state as indicated by the instruction.

Notes

1 A reserved operand fault occurs if **pos** GTRU 31 and the bit is contained in a register.

2 On a reserved operand fault, the field is unaffected, and the condition codes are UNPREDICTABLE.

3 The modification of the bit is not an interlocked operation. See BBSSI and BBCCI for interlocking instructions.

BB

Branch on Bit Interlocked

FORMAT *opcode pos.rl, base.vb, displ.bb*

condition codes

N	$\leftarrow$ N;
Z	$\leftarrow$ Z;
V	$\leftarrow$ V;
C	$\leftarrow$ C;

exceptions

reserved operand

opcodes

E6	BBSSI	Branch on Bit Set and Set Interlocked
E7	BBCCI	Branch on Bit Clear and Clear Interlocked

DESCRIPTION

The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result. Regardless of whether the branch is taken, the tested bit is put in the new state as indicated by the instruction. If the bit is contained in memory, the reading of the state of the bit and the setting of the bit to the new state is an interlocked operation. No other processor or I/O device can do an interlocked access on this bit during the interlocked operation.

Notes

1. A reserved operand fault occurs if **pos** GTRU 31 and the specified bit is contained in a register.
2. On a reserved operand fault, the field is unaffected, and the condition codes are UNPREDICTABLE.
3. Except for memory interlocking, BBSSI is equivalent to BBSS, and BBCCI is equivalent to BBCC.
4. This instruction is designed to modify interlocks with other processors or devices. For example, to implement "busy waiting":

```
1$:     BBSSI   bit,base,1$
```

BLB

Branch on Low Bit

FORMAT

opcode *src.rl, displ.bb*

condition codes

N	←	N;
Z	←	Z;
V	←	V;
C	←	C;

exceptions

None.

opcodes

E8	BLBS	Branch on Low Bit Set
E9	BLBC	Branch on Low Bit Clear

DESCRIPTION

The low bit (bit 0) of the source operand is tested. If it is equal to the test state indicated by the instruction, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result.

BR

Branch

FORMAT *opcode* *displ.bx*

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

None.

opcodes

11	BRB	Branch with Byte Displacement
31	BRW	Branch with Word Displacement

DESCRIPTION

The sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result.

BSB

Branch to Subroutine

FORMAT

opcode *displ.bx*

condition codes

N	$\leftarrow$ N;
Z	$\leftarrow$ Z;
V	$\leftarrow$ V;
C	$\leftarrow$ C;

exceptions

None.

opcodes

10	BSBB	Branch to Subroutine with Byte Displacement
30	BSBW	Branch to Subroutine with Word Displacement

DESCRIPTION

The program counter (PC) is pushed on the stack as a longword. The sign-extended branch displacement is added to the PC, and the PC is replaced by the result.

CASE

Case

FORMAT

opcode *selector.rx, base.rx, limit.rx, displ[0].bw, ..., displ[limit].bw*

condition codes

N	⟵	tmp LSS limit;
Z	⟵	tmp EQL limit;
V	⟵	0;
C	⟵	tmp LSSU limit;

exceptions

None.

opcodes

8F	CASEB	Case Byte
AF	CASEW	Case Word
CF	CASEL	Case Long

DESCRIPTION

The base operand is subtracted from the selector operand, and the result replaces a temporary operand. The temporary operand is compared with the limit operand; if it is less than or equal unsigned, a branch displacement selected by the temporary value is added to the program counter (PC), and the PC is replaced by the result. Otherwise, twice the sum of the limit operand and 1 is added to the PC, and the PC is replaced by the result. This operation causes the PC to be moved past the array of branch displacements. Regardless of the branch taken, the condition codes are modified as a result of the comparison of the temporary operand with the limit operand.

Notes

1 After operand evaluation, the PC points at **displ**[0], not to the next instruction. The branch displacements are relative to the address of **displ**[0].

2 The selector and base operands can both be considered as either signed or unsigned integers.

In the following example, the CASEB instruction selects one of eight displacements immediately following the instruction. The example is for illustration only. An actual instruction would use run-time variables instead of the assembly-time static values shown. Also, in an actual instruction, the displacements selected by the CASEB instruction would be branches to various routines.

```
        .PSECT    CODE, PIC, SHR, WRT, EXE, LONG
TABIND: .WORD 4
        .ENTRY    START,^M<>
        CLRW      R4
        CLRW      R5
        MOVW      #0,R4
        MOVW      #7,R5
        CASEB     TABIND,R4,R5
TAB:    .WORD     1$-TAB
        .WORD     2$-TAB
        .WORD     3$-TAB
        .WORD     4$-TAB
        .WORD     5$-TAB
        .WORD     6$-TAB
        .WORD     7$-TAB
        BRB       9$
1$:     .ASCII    /AT 1/
2$:     .ASCII    /AT 2/
3$:     .ASCII    /AT 3/
4$:     .ASCII    /AT 4/
5$:     .ASCII    /AT 5/
6$:     .ASCII    /AT 6/
7$:     .ASCII    /AT 7/
8$:     .ASCII    /AT 8/
9$:     $EXIT_S
        .END START
```

The objective of the CASE instruction is to transfer control to one of many possible locations depending on the value of "selector," or TABIND, as shown in the example. These locations are labeled in the example from 1$: to 8$:.

In the example, the table contains eight branch displacements. In all cases, the limit operand (here shown as R5, which contains a 7) is one less than the number of displacements (8) in the table. The base operand (here shown as R4, which contains a zero) is the lowest permissible value for TABIND.

The CASE instruction subtracts base (contents of R4, a zero) from the value of TABIND to produce a zero-origin index into the table. The limit (contents of R5, a 7) is compared with this index to ensure that the table limit is not exceeded.

After operand evaluation, the program counter (PC) points to TAB:. The locations to which branching occurs are represented in the table as displacements. The displacement in the table selected by TABIND is added to the PC to form a destination address. The destination selected in the example is at location 5$:. In practical usage, this location would contain a branch to a specific routine.

JMP

Jump

FORMAT

opcode *dst.ab*

condition codes

N	←	N;
Z	←	Z;
V	←	V;
C	←	C;

exceptions

None.

opcodes

17	JMP	Jump

DESCRIPTION

The program counter (PC) is replaced by the destination operand.

JSB

Jump to Subroutine

FORMAT *opcode dst.ab*

condition codes

N	←	N;
Z	←	Z;
V	←	V;
C	←	C;

exceptions

None.

opcodes

16	JSB	Jump to Subroutine

DESCRIPTION

The program counter (PC) is pushed onto the stack as a longword. The PC is replaced by the destination operand.

Note

Because the operand specifier conventions cause the evaluation of the destination operand before saving the PC, you can use JSB for coroutine calls with the stack used for linkage. The form of this call is:

```
JSB @(SP)+
```

RSB

Return from Subroutine

FORMAT

opcode

condition codes

N	$\leftarrow$ N;
Z	$\leftarrow$ Z;
V	$\leftarrow$ V;
C	$\leftarrow$ C;

exceptions

None.

opcodes

05	RSB	Return from Subroutine

DESCRIPTION

The program counter (PC) is replaced by a longword popped from the stack.

Notes

1 Use RSB to return from subroutines called by the BSBB, BSBW, and JSB instructions.

2 RSB is equivalent to JMP @(SP)+, but is 1 byte shorter.

SOBGEQ

Subtract One and Branch Greater Than or Equal

FORMAT

opcode index.ml, displ.bb

condition codes

N	⟵ index LSS 0;
Z	⟵ index EQL 0;
V	⟵ {integer overflow};
C	⟵ C;

exceptions

integer overflow

opcodes

F4	SOBGEQ	Subtract One and Branch Greater Than or Equal

DESCRIPTION

One is subtracted from the index operand, and the index operand is replaced by the result. If the index operand is greater than or equal to zero, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result.

Notes

1. Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow, the index operand is replaced by the largest positive integer; therefore, the branch is taken.
2. The C-bit is unaffected.

SOBGTR

Subtract One and Branch Greater Than

FORMAT

opcode index.ml, displ.bb

condition codes

N	$\longleftarrow$ index LSS 0;
Z	$\longleftarrow$ index EQL 0;
V	$\longleftarrow$ {integer overflow};
C	$\longleftarrow$ C;

exceptions

integer overflow

opcodes

F5	SOBGTR	Subtract One and Branch Greater Than

DESCRIPTION

One is subtracted from the index operand, and the index operand is replaced by the result. If the index operand is greater than zero, the sign-extended branch displacement is added to the program counter (PC), and the PC is replaced by the result.

Notes

1. Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow, the index operand is replaced by the largest positive integer, and thus, the branch is taken.
2. The C-bit is unaffected.

9.2.5 Procedure Call Instructions

The following three instructions implement a standard procedure calling interface:

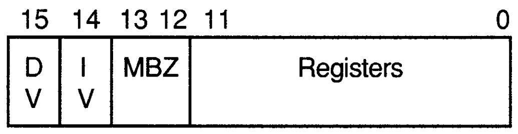
- CALLG
- CALLS
- RET

CALLG and CALLS call the procedure. The RETURN instruction returns from the procedure. Refer to the *Introduction to VMS System Routines* for the procedure calling standard.

The CALLG instruction calls a procedure with the argument list in an arbitrary location.

The CALLS instruction calls a procedure with the argument list on the stack. Upon return after a CALLS instruction, this list is automatically removed from the stack. Both call instructions specify the address of the entry point of the procedure being called. The entry point is assumed to consist of a word called the *entry mask* followed by the procedure's instructions. The procedure terminates by executing a RET instruction.

The entry mask specifies the register use and overflow enables of the subprocedure.



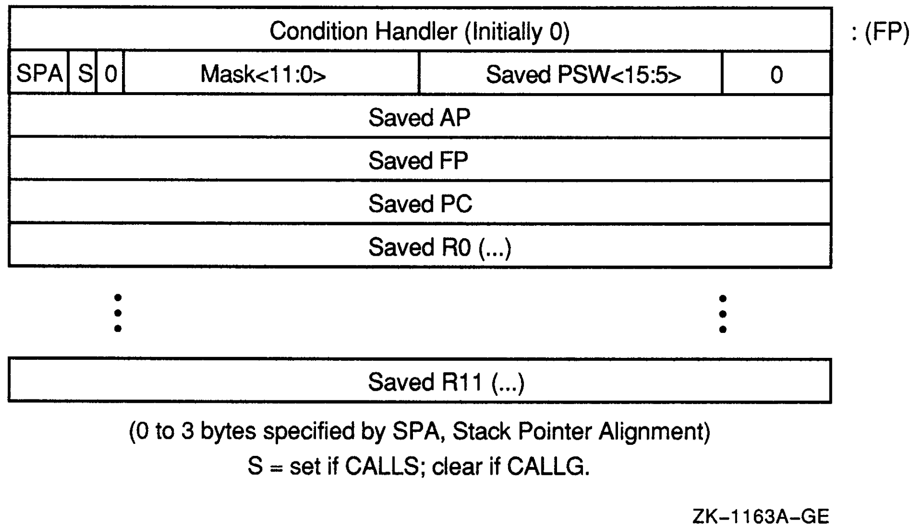
ZK-1162A-GE

At the occurrence of one of the call instructions, the stack is aligned to a longword boundary, and the trap enables in the processor status longword (PSW) are set to a known state to ensure consistent behavior of the called procedure. Integer overflow enable and decimal overflow enable are affected according to bits 14 and 15 of the entry mask, respectively. Floating underflow enable is cleared. Registers R11 to R0, specified by bits 11 to 0, respectively, are saved on the stack and are restored by the RET instruction. In addition, the program counter (PC), stack pointer (SP), frame pointer (FP), and argument pointer (AP) are always preserved by the CALL instructions and restored by the RET instruction.

All external procedure calls generated by standard Digital language processors and all intermodule calls to major VAX software subsystems comply with the procedure calling software standard (see the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*). The procedure calling standard requires that all registers in the range R2 to R11 used in the procedure must appear in the mask. R0 and R1 are not preserved by any called procedure that complies with the procedure calling standard.

To preserve the state, the CALL instructions form a structure on the stack termed a **call frame** or **stack frame**. The call frame contains the saved registers, the saved PSW, the register save mask, and several control bits. The frame also includes a longword that the CALL instructions clear. The system uses this longword to implement the VMS condition handling facility (see the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*). At the end of execution of the CALL instruction,the frame pointer (FP) contains the address of the stack frame. The RET instruction uses the contents of FP to find the stack frame and the restore state. The condition handling facility assumes that FP always points to the stack frame.

The stack frame has the following format:


Condition Handler (Initially 0)
: (FP)
SPA
S
0
Mask<11:0>
Saved PSW<15:5>
0
Saved AP
Saved FP
Saved PC
Saved R0 (...)
Saved R11 (...)
(0 to 3 bytes specified by SPA, Stack Pointer Alignment)
S = set if CALLS; clear if CALLG.
ZK-1163A-GE

Note that the saved condition codes and the saved trace enable (PSW<T>) are cleared.

The contents of the frame PSW<3:0> at the time RET is executed will become the condition codes resulting from the execution of the procedure. Similarly, the content of the frame PSW<4> at the time the RET is executed will become the PSW<T> bit.

The following instructions are described in this section.

	Description and Opcode	Number of Instructions
1.	Call Procedure with General Argument List CALLG arglist.ab, dst.ab, {-(SP).w*}	1
2.	Call Procedure with Stack Argument List CALLS numarg.rl, dst.ab, {-(SP).w*}	1
3.	Return from Procedure RET {(SP)+.r*}	1

CALLG

Call Procedure with General Argument List

FORMAT

opcode arglist.ab, dst.ab

condition codes

N	⟵ 0;
Z	⟵ 0;
V	⟵ 0;
C	⟵ 0;

exceptions

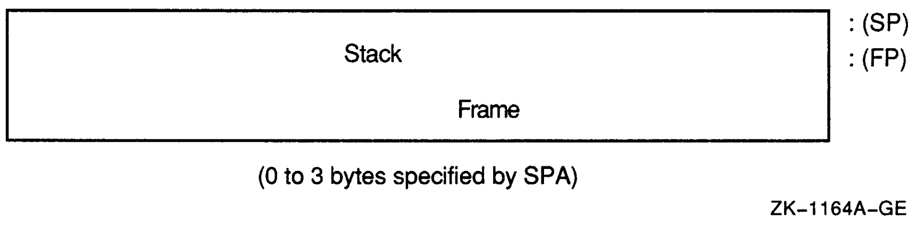
reserved operand

opcodes

FA	CALLG	Call Procedure with General Argument List

DESCRIPTION

The stack pointer (SP) is saved in a temporary register. Bits 1:0 are replaced by zero, so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit 0, and the contents of registers whose numbers correspond to set bits in the mask are pushed on the stack as longwords. The program counter (PC), frame pointer (FP), and argument pointer (AP) are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved low 2 bits of the SP in bits 31:30, a zero in bits 29 and 28, the low 12 bits of the procedure entry mask in bits 27:16, and the processor status word (PSW) in bits 15:0 with T cleared are pushed on the stack. A longword zero is pushed on the stack. The FP is replaced by the SP. The AP is replaced by the **arglist** operand. The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask, respectively; floating underflow is cleared. The T-bit is unaffected. The PC is replaced by the sum of destination operand plus 2, which transfers control to the called procedure at the byte beyond the entry mask.



ZK−1164A−GE

Notes

1 If bits 13:12 of the entry mask are not zero, a reserved operand fault occurs.

2 On a reserved operand fault, condition codes are UNPREDICTABLE.

3 The procedure calling standard and the condition handling facility require the following register saving conventions:

- R0 and R1 are always available for function return values and are never saved in the entry mask.
- All registers R2 to R11 that are modified in the called procedure must be preserved in the mask.

Refer to the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*.

CALLS

Call Procedure with Stack Argument List

FORMAT *opcode numarg.rl, dst.ab*

condition codes

N ⟵ 0;
Z ⟵ 0;
V ⟵ 0;
C ⟵ 0;

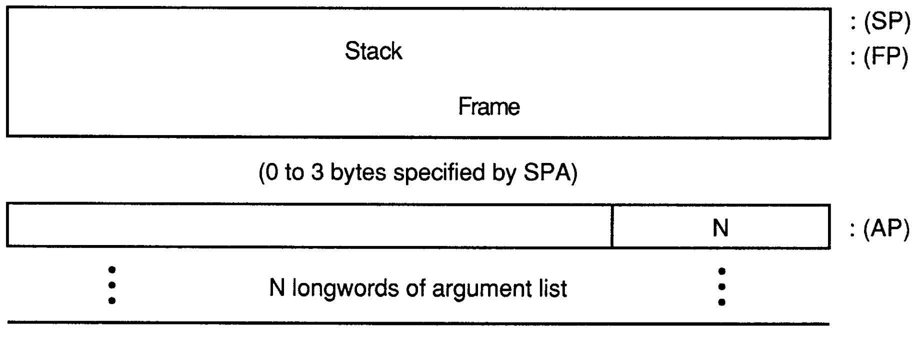
exceptions

reserved operand

opcodes

FB	CALLS	Call Procedure with Stack Argument List

DESCRIPTION The **numarg** operand is pushed on the stack as a longword (byte 0 contains the number of arguments; Digital software uses the high-order 24 bits). The stack pointer (SP) is saved in a temporary register, and then bits 1:0 of the SP are replaced by zero so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit 0, and the contents of registers whose numbers correspond to set bits in the mask are pushed on the stack. The program counter (PC), frame pointer (FP), and argument pointer (AP) are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved low 2 bits of the SP in bits 31:30, a 1 in bit 29, a zero in bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the processor status word (PSW) in bits 15:0 with T cleared is pushed on the stack. A longword zero is pushed on the stack. The FP is replaced by the SP. The AP is set to the value of the stack pointer after the **numarg** operand was pushed on the stack. The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask, respectively. Floating underflow is cleared. T-Bit is unaffected. The PC is replaced by the sum of destination operand plus 2, which transfers control to the called procedure at the byte beyond the entry mask. The appearance of the stack after CALLS is executed is:



Notes

1 If bits 13:12 of the entry mask are not zero, a reserved operand fault occurs.

2 On a reserved operand fault, the condition codes are UNPREDICTABLE.

3 Normal use is to push the **arglist** onto the stack in reverse order prior to the CALLS. On return, the **arglist** is removed from the stack automatically.

4 The procedure calling standard and the condition handling facility require the following register saving conventions:

- R0 and R1 are always available for function return values and are never saved in the entry mask.
- All registers R2 to R11 that are modified in the called procedure must be preserved in the entry mask. Refer to the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*.

RET

Return from Procedure

FORMAT

opcode

condition codes

```
N  ← tmp1<3>;
Z  ← tmp1<2>;
V  ← tmp1<1>;
C  ← tmp1<0>;
```

exceptions

reserved operand

opcodes

04	RET	Return from Procedure

DESCRIPTION

The stack pointer (SP) is replaced by the frame pointer (FP) plus 4. A longword containing stack alignment bits in bits 31:30, a CALLS/CALLG flag in bit 29, the low 12 bits of the procedure entry mask in bits 27:16, and a saved processor status word (PSW) in bits 15:0 is popped from the stack and saved in a temporary. The program counter (PC), frame pointer (FP), and argument pointer (AP) are replaced by longwords popped from the stack. A register restore mask is formed from bits 27:16 of the temporary. Scanning from bit 0 to bit 11 of the restore mask, the contents of registers whose numbers are indicated by set bits in the mask are replaced by longwords popped from the stack. The SP is incremented by 31:30 of the temporary. The PSW is replaced by bits 15:0 of the temporary. If bit 29 in the temporary is 1 (indicating that the procedure was called by CALLS), a longword containing the number of arguments is popped from the stack. Four times the unsigned value of the low byte of this longword is added to the SP, and the SP is replaced by the result.

Notes

1. A reserved operand fault occurs if **tmp1**<15:8> NEQ 0.
2. On a reserved operand fault, the condition codes are UNPREDICTABLE.
3. The value of **tmp1**<28> is ignored.
4. The procedure calling standard and condition handling facility assume that procedures which return a function value or a status code do so in R0, or R0 and R1. Refer to the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*.

9.2.6 Miscellaneous Instructions

The following instructions are described in this section.

	Description and Opcode	Number of Instructions
1.	Bit Clear PSW BICPSW mask.rw	1
2.	Bit Set PSW BISPSW mask.rw	1
3.	Breakpoint Fault BPT {-(KSP).w*}	1
4.	Halt HALT {-(KSP).w*}	1
5.	Index INDEX subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl	1
6.	Move from PSL MOVPSL dst.wl	1
7.	No Operation NOP	1
8.	Pop Registers POPR mask.rw, {(SP)+.r*}	1
9.	Push Registers PUSHR mask.rw, {-(SP).w*}	1
10.	Extended Function Call XFC {unspecified operands}	1

BICPSW

Bit Clear PSW

FORMAT

opcode *mask.rw*

condition codes

N	⟵ N AND {NOT mask<3>};
Z	⟵ Z AND {NOT mask<2>};
V	⟵ V AND {NOT mask<1>};
C	⟵ C AND {NOT mask<0>};

exceptions

reserved operand

opcodes

B9	BICPSW	Bit Clear PSW

DESCRIPTION

The result of the logical AND on processor status word (PSW) and the one's complement of the mask operand replaces PSW.

Note

A reserved operand fault occurs if **mask**<15:8> is not zero. On a reserved operand fault, the PSW is not affected.

BISPSW

Bit Set PSW

FORMAT

opcode *mask.rw*

condition codes

N	⟵ N OR mask<3>;
Z	⟵ Z OR mask<2>;
V	⟵ V OR mask<1>;
C	⟵ C OR mask<0>;

exceptions

reserved operand

opcodes

B8	BISPSW	Bit Set PSW

DESCRIPTION

The result of the logical OR on processor status word (PSW) and the mask operand replaces PSW.

Note

A reserved operand fault occurs if **mask**<15:8> is not zero. On a reserved operand fault, the PSW is not affected.

BPT

Breakpoint Fault

FORMAT

opcode

condition codes

```
N   ←— 0; ! Condition codes cleared after BPT fault
Z   ←— 0;
V   ←— 0;
C   ←— 0;
```

exceptions

None.

opcodes

03	BPT	Breakpoint Fault

DESCRIPTION

To understand the operation of this instruction, refer to Appendix E. This instruction, together with the T-bit, is used to implement debugging facilities.

HALT

Halt

FORMAT *opcode*

condition codes

```
N  ← 0; ! If privileged instruction fault,
Z  ← 0; ! condition codes are cleared after
V  ← 0; ! the fault. PSL saved on stack
C  ← 0; ! contains condition codes prior to HALT.

N  ← N; ! If processor halt
Z  ← Z;
V  ← V;
C  ← C;
```

exceptions

privileged instruction

opcodes

00	HALT	Halt

DESCRIPTION

If the process is running in kernel mode, the processor is halted. Otherwise, a privileged instruction fault occurs. For information about privileged instruction faults, refer to Appendix E.

Note

This opcode is zero to trap many branches to data.

INDEX

Compute Index

FORMAT

opcode *subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl*

condition codes

N	← indexout LSS 0;
Z	← indexout EQL 0;
V	← 0;
C	← 0;

exceptions

subscript range

opcodes

0A	INDEX	index

DESCRIPTION

The **indexin** operand is added to the **subscript** operand and the sum multiplied by the **size** operand. The **indexout** operand is replaced by the result. If the **subscript** operand is less than the **low** operand or greater than the **high** operand, a subscript range trap is taken.

Notes

1. No arithmetic exception other than subscript range can result from this instruction. Therefore, no indication is given if overflow occurs in either the add or the multiply steps. If overflow occurs on the add step, the sum is the low-order 32 bits of the true result. If overflow occurs on the multiply step, the **indexout** operand is replaced by the low-order 32 bits of the true product of the sum and the **subscript** operand. In the normal use of this instruction, overflow cannot occur without a subscript range trap occurring.
2. The index instruction is useful in index calculations for arrays of the fixed-length data types (integer and floating) and for index calculations for arrays of bit fields, character strings, and decimal strings. The **indexin** operand permits cascading INDEX instructions for multidimensional arrays. For one-dimensional bit field arrays, it also permits introduction of the constant portion of an index calculation that is not readily absorbed by address arithmetic. The following notes show some of the uses of INDEX.

3 The following example shows a sequence of COBOL statements and the VAX MACRO code their compilation might generate:

```
COBOL:

01  A-ARRAY.
    02  A PIC X(10) OCCURS 15 TIMES.

01  B PIC X(10).
    MOVE A(I) TO B.

MACRO:

INDEX I, #1, #15, #10, #0, R0

MOVC3 #10, A-10[R0], B.
```

4 The following example shows a sequence of PL/I statements and the VAX MACRO code their compilation might generate:

```
PL/I:

DCL A(-3:10) BIT (5);
A(I) = 1;

MACRO:

INDEX I, #-3, #10, #5, #3, R0

INSV  #1, R0, #5, A ; Assumes A is byte aligned
```

5 The following example shows a sequence of FORTRAN statements and the VAX MACRO code their compilation might generate:

```
FORTRAN:

INTEGER*4 A(L1:U1, L2:U2), I, J
A(I,J) = 1

MACRO:

INDEX J, #L2, #U2, #M1, #0, R0; M1=U1-L1+1
INDEX I, #L1, #U1, #1, R0, R0;
MOVL     #1, A-a[R0]; a = {{L2*M1} + L1} *4
```

MOVPSL

Move from PSL

FORMAT

opcode *dst.wl*

condition codes

N	⟵ N;
Z	⟵ Z;
V	⟵ V;
C	⟵ C;

exceptions

None.

opcodes

DC	MOVPSL	Move from PSL

DESCRIPTION

The destination operand is replaced by processor status longword (PSL).

NOP

No Operation

FORMAT

opcode

condition codes

N	← N;
Z	← Z;
V	← V;
C	← C;

exceptions

None.

opcodes

01	NOP	No Operation

DESCRIPTION

No operation is performed. Because the time delay caused by a NOP instruction is dependent on processor type, Digital recommends that you do not use NOP as a means of delaying program execution. When you must have a program wait for a specified period, you should use a macro, such as the TIMEDWAIT macro, or code sequence that is not dependent on the processor's internal speed.

POPR

Pop Registers

FORMAT

opcode *mask.rw*

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

None.

opcodes

BA	POPR	Pop Registers

DESCRIPTION

The contents of registers whose numbers correspond to set bits in the mask operand are replaced by longwords popped from the stack. R[n] is replaced if **mask**<n> is set. The mask is scanned from bit 0 to bit 14. Bit 15 is ignored.

PUSHR

Push Registers

FORMAT

opcode *mask.rw*

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

None.

opcodes

BB	PUSHR	Push Registers

DESCRIPTION

The contents of registers whose numbers correspond to set bits in the mask operand are pushed on the stack as longwords. R[n] is pushed if **mask**<n> is set. The mask is scanned from bit 14 to bit 0. Bit 15 is ignored.

Note

The order of pushing is specified so that the contents of higher-numbered registers are stored at higher memory addresses. An example of a result of this would be a double-floating datum stored in adjacent registers being stored by PUSHR in memory in the correct order.

XFC

Extended Function Call

FORMAT

opcode

condition codes

N ← 0;
Z ← 0;
V ← 0;
C ← 0;

exceptions

None.

opcodes

FC	XFC	Extended Function Call

DESCRIPTION

To understand the operation of this instruction, refer to Appendix E and the *VAX Architecture Reference Manual*. This instruction provides for customer-defined extensions to the instruction set.

9.2.7 Queue Instructions

A queue is a circular, doubly linked list. A queue entry is specified by its address. Each queue entry is linked to the next by a pair of longwords. The first longword is the forward link; it specifies the location of the succeeding entry. The second longword is the backward link; it specifies the location of the preceding entry. Because a queue contains redundant links, it is possible to create ill-formed queues. The VAX instructions produce UNPREDICTABLE results when used on ill-formed queues.

A queue is classified by the type of link that it uses. The VAX supports two distinct types of links: absolute and self-relative.

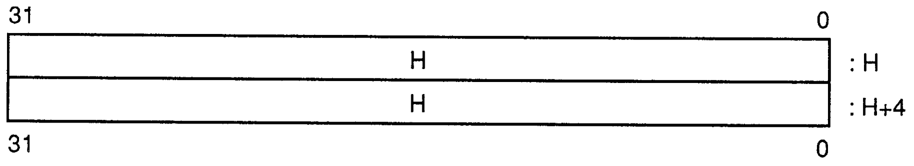
9.2.7.1 Absolute Queues

Absolute queues use absolute addresses as links. Queue entries are linked by a pair of longwords. The first (lowest-addressed) longword is the forward link; it is the address of the succeeding queue entry. The second (highest-addressed) longword is the backward link; it is the address of the preceding queue entry.

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

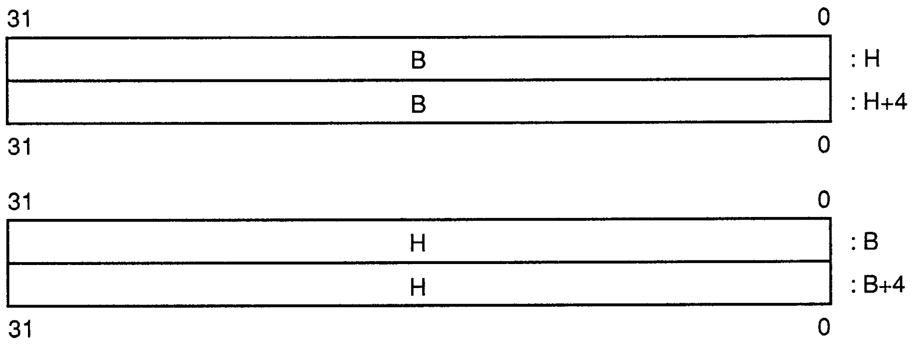
Two general operations can be performed on queues: insertion of entries and removal of entries. Generally, entries can be inserted or removed only at the head or tail of a queue. (Under certain restrictions they can be inserted or removed elsewhere; this is discussed later.)

The following text contains examples of queue operations. An empty queue is specified by its header at address H.



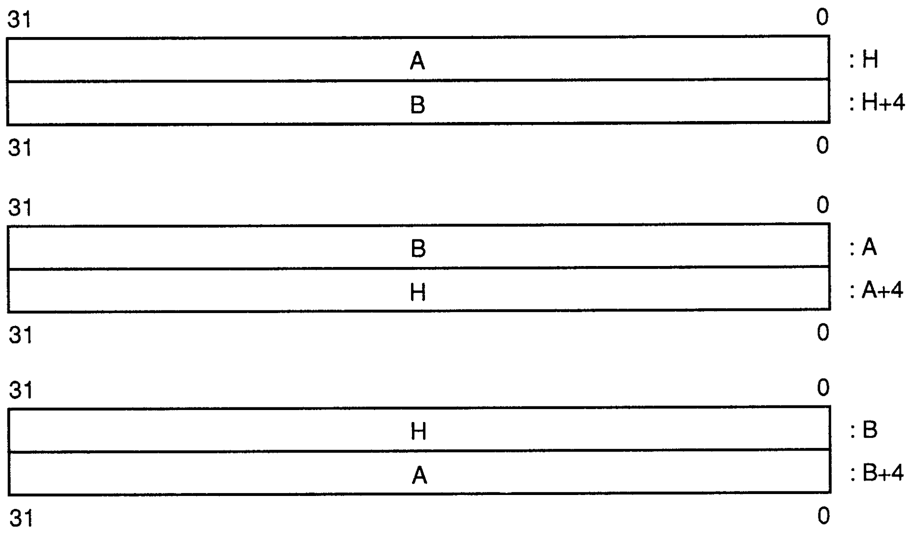
ZK-1166A-GE

If an entry at address B is inserted into an empty queue (at either the head or the tail), the queue appears as follows:



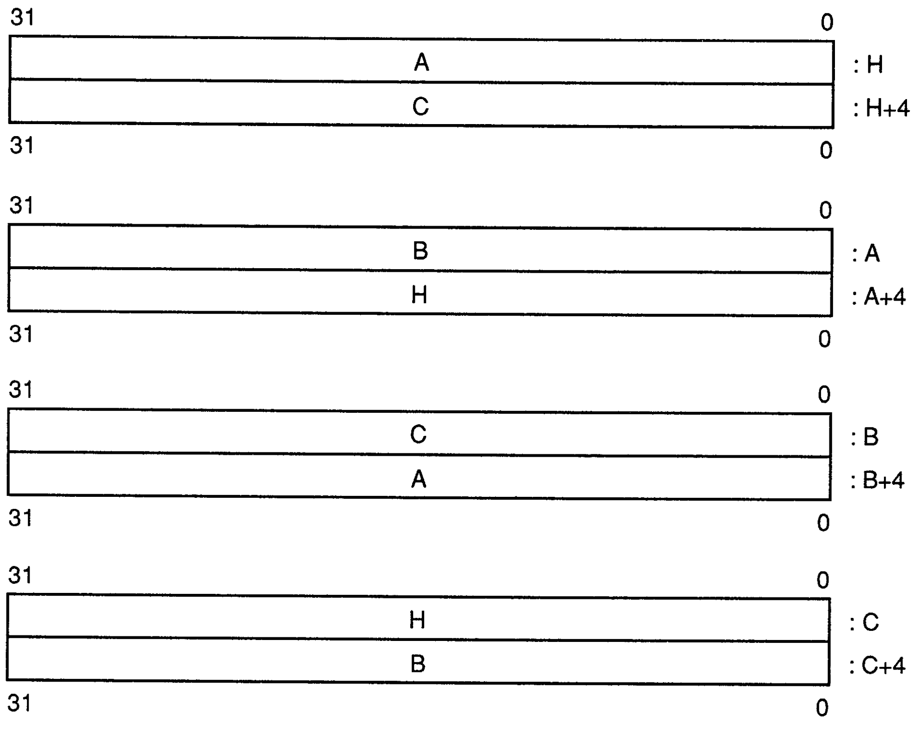
ZK-1167A-GE

If an entry at address A is inserted at the head of the queue, the queue appears as follows:



ZK-1168A-GE

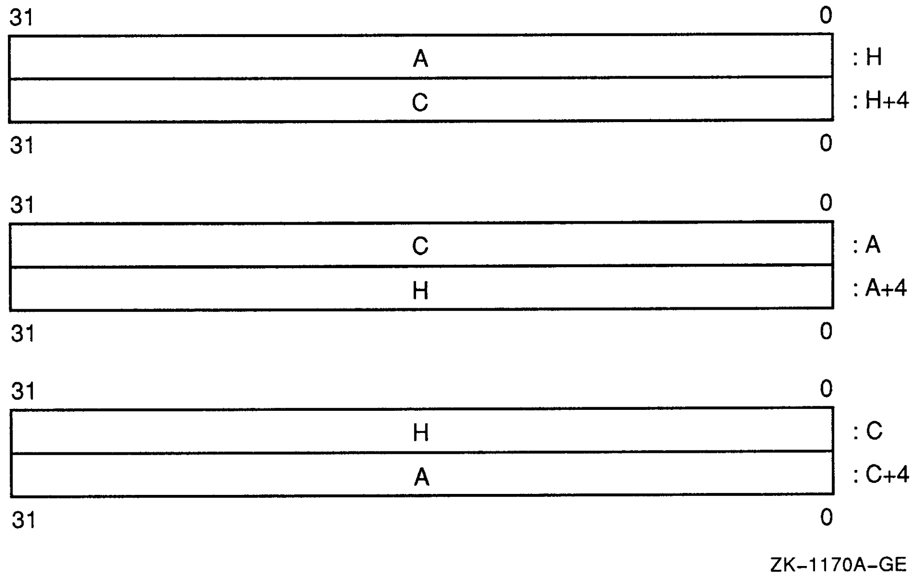
Finally, if an entry at address C is inserted at the tail, the queue appears as follows:



ZK-1169A-GE

Following the preceding steps in reverse order gives the effect of removal at the tail and removal at the head.

If more than one process can perform operations on a queue simultaneously, insertions and removals should only be done at the head or tail of the queue. If only one process (or one process at a time) can perform operations on a queue, insertions and removals can be made at other than the head or tail of the queue. In the preceding example with the queue containing entries A, B, and C, the entry at address B can be removed, giving the following:



The reason for this restriction is that operations at the head or tail are always valid because the queue header is always present. Operations elsewhere in the queue depend on specific entries being present and may become invalid if another process is simultaneously performing operations on the queue.

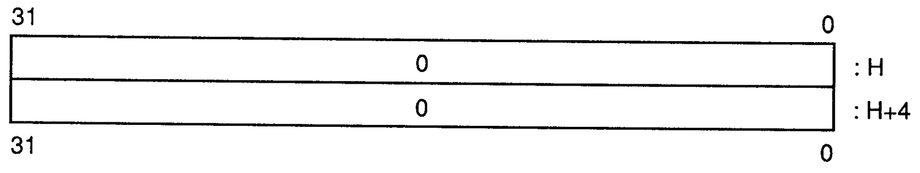
Two instructions are provided for manipulating absolute queues: INSQUE and REMQUE. INSQUE inserts an entry specified by an entry operand into the queue following the entry specified by the predecessor operand. REMQUE removes the entry specified by the entry operand. Queue entries can be on arbitrary byte boundaries. Both INSQUE and REMQUE are implemented as noninterruptible instructions.

9.2.7.2 Self-Relative Queues

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

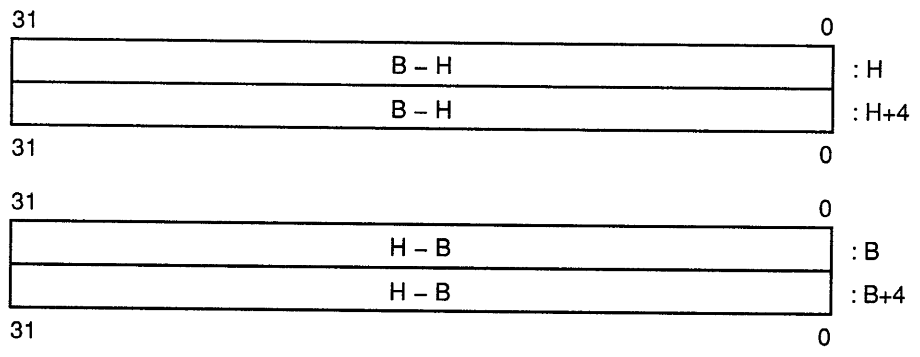
A queue is specified by a queue header, which also consists of two longword links. The forward link of the header is the address of the entry called the *head* of the queue. The backward link of the header is the address of the entry called the *tail* of the queue. The forward link of the tail points to the header.

The following text contains examples of queue operations. An empty queue is specified by its header at address H. Because the queue is empty, the self-relative links must be zero, as shown.



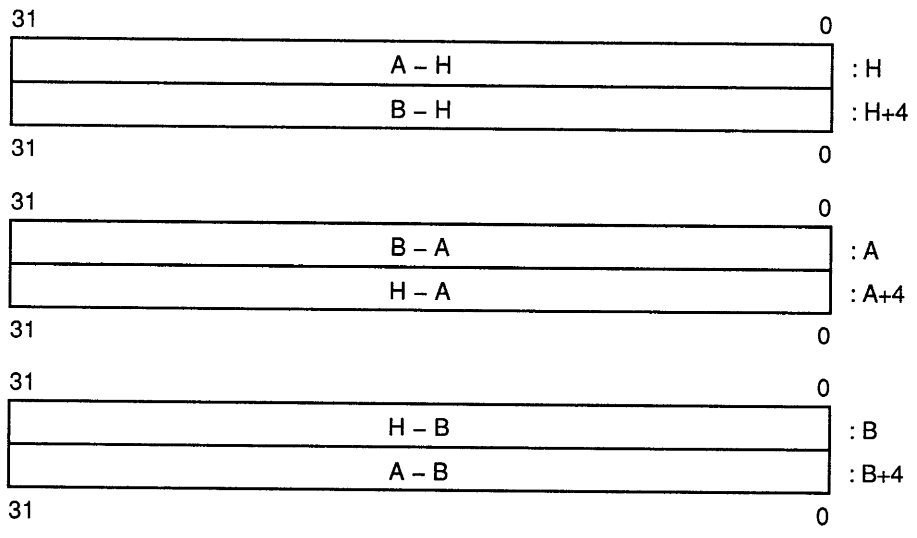
ZK-1171A-GE

If an entry at address B is inserted into an empty queue (at either the head or tail), the queue appears as follows:



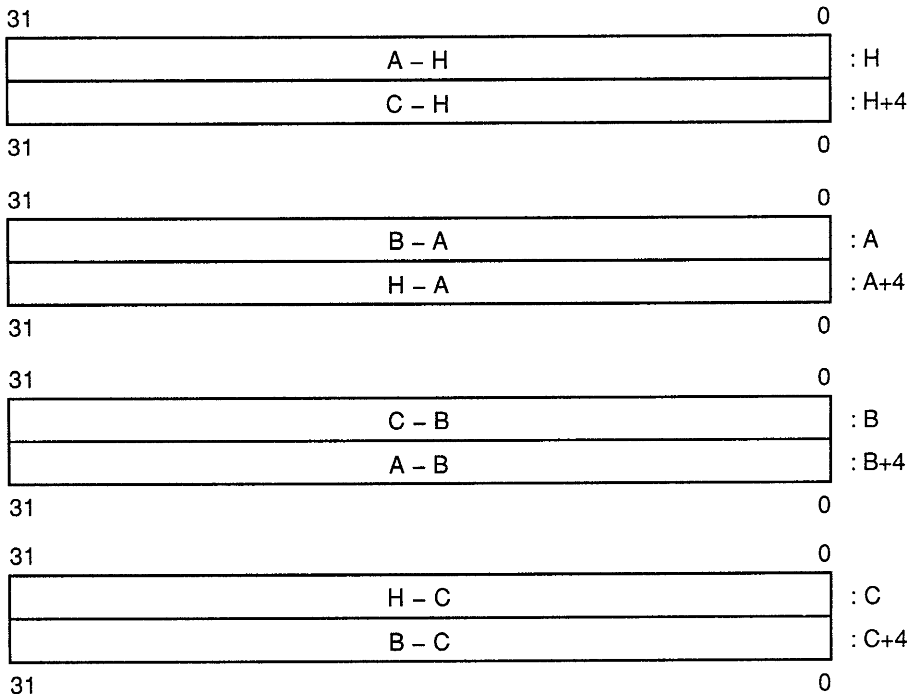
ZK-1172A-GE

If an entry at address A is inserted at the head of the queue, the queue appears as follows:



ZK-1173A-GE

Finally, if an entry at address C is inserted at the tail, the queue appears as follows:



ZK-1174A-GE

Following the previous steps in reverse order gives the effect of removal at the tail and at the head.

The following four instructions manipulate self-relative queues:

1 INSQHI—Insert entry into queue at head, interlocked.

2 INSQTI—Insert entry into queue at tail, interlocked.

3 REMQHI—Remove entry from queue at head, interlocked.

4 REMQTI—Remove entry from queue at tail, interlocked.

These operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared list without additional synchronization. Queue entries must be quadword aligned. A hardware-supported interlocked memory access mechanism is used to read the queue header. Bit 0 of the queue header is used as a secondary interlock; it is set when the queue is being accessed. If an interlocked queue instruction encounters the secondary interlock set, it terminates after setting the condition codes to indicate failure to gain access to the queue. If the secondary interlock bit is not set, then the interlocked queue instruction sets it during its operation and clears it at instruction completion. In this way, other interlocked queue instructions are prevented from operating on the same queue.

9.2.7.3 Instruction Descriptions

The following instructions are described in this section:

	Description and Opcode	Number of Instructions
1.	Insert Entry into Queue at Head, Interlocked INSQHI entry.ab, header.aq	1
2.	Insert Entry into Queue at Tail, Interlocked INSQTI entry.ab, header.aq	1
3.	Insert Entry in Queue INSQUE entry.ab, pred.ab	1
4.	Remove Entry from Queue at Head, Interlocked REMQHI header.aq, addr.wl	1
5.	Remove Entry from Queue at Tail, Interlocked REMQTI header.aq, addr.wl	1
6.	Remove Entry from Queue REMQUE entry.ab, addr.wl	1

INSQHI

Insert Entry into Queue at Head, Interlocked

FORMAT

opcode entry.ab, header.aq

condition codes

```
if {insertion succeeded} then
     begin
     N ⟵ 0;
     Z ⟵ (entry) EQL (entry+4);          ! First entry in queue
     V ⟵ 0;
     C ⟵ 0;
     end;
else
     begin
     N ⟵ 0;
     Z ⟵ 0;
     V ⟵ 0;
     C ⟵ 1;                              ! Secondary interlock failed
     end;
```

exceptions

reserved operand

opcodes

5C	INSQHI	Insert Entry into Queue at Head, Interlocked

DESCRIPTION

The entry specified by the entry operand is inserted into the queue following the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This method ensures that if a memory management exception occurs (see Appendix E), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

Notes

1 Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

2 The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3 To set a software interlock realized with a queue, you can use the following:

```
INSERT:
        INSQHI   ...              ; Was queue empty?
        BEQL     1$               ; Yes
        BCS      INSERT           ; Try inserting again
        CALL     WAIT(...)        ; No, wait
1$:
```

4 During access validation, any access that cannot be completed results in a memory management exception even though the queue insertion is not started.

5 A reserved operand fault occurs if **entry** or **header** is an address that is not quadword aligned (that is, <2:0> NEQU 0) or if **header**<2:1> is not zero. A reserved operand fault also occurs if **header** equals **entry**. In this case, the queue is not altered.

INSQTI

Insert Entry into Queue at Tail, Interlocked

FORMAT

opcode *entry.ab, header.aq*

condition codes

```
if {insertion succeeded} then
     begin
     N ⟵ 0;
     Z ⟵ (entry) EQL (entry+4);          ! First entry in queue
     V ⟵ 0;
     C ⟵ 0;
     end;
else
     begin
     N ⟵ 0;
     Z ⟵ 0;
     V ⟵ 0;
     C ⟵ 1;                              ! Secondary interlock failed
     end;
```

exceptions

reserved operand

opcodes

5D	INSQTI	Insert Entry into Queue at Tail, Interlocked

DESCRIPTION

The entry specified by the entry operand is inserted into the queue preceding the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise, it is cleared. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This method ensures that if a memory management exception occurs (see Appendix E), queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

Notes

1 Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

2 The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3 To set a software interlock realized with a queue, you can use the following:

```
INSERT:
        INSQHI  ...             ; Was queue empty?
        BEQL    1$              ; Yes
        BCS     INSERT          ; Try inserting again
        CALL    WAIT(...)       ; No, wait
1$:
```

4 During access validation, any access that cannot be completed results in a memory management exception even though the queue insertion is not started.

5 A reserved operand fault occurs if **entry**, **header**, or (**header**+4) is an address that is not quadword aligned (that is, <2:0> NEQU 0) or if **header**<2:1> is not zero. A reserved operand fault also occurs if **header** equals **entry**. In this case, the queue is not altered.

INSQUE

Insert Entry in Queue

FORMAT

opcode entry.ab, pred.ab

condition codes

N	⟵ (entry) LSS (entry+4);
Z	⟵ (entry) EQL (entry+4); ! First entry in queue
V	⟵ 0;
C	⟵ (entry) LSSU (entry+4);

exceptions

None.

opcodes

0E	INSQUE	Insert Entry in Queue

DESCRIPTION

The entry specified by the entry operand is inserted into the queue following the entry specified by the predecessor operand. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This method ensures that if a memory management exception occurs (see Appendix E), the queue is left in a consistent state.

Notes

1 The following three types of insertion can be performed by appropriate choice of the predecessor operand:

- Insert at head:

```
INSQUE  entry, h          ; h is queue head
```

- Insert at tail:

```
INSQUE  entry,@h+4        ; h is queue head
(Note "@" in this case only)
```

- Insert after arbitrary predecessor:

```
INSQUE  entry,p           ; p is predecessor
```

2 Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

3 The INSQUE and REMQUE instructions are implemented such that cooperating software processes in a single processor may access a shared list without additional synchronization, if the insertions and removals are only at the head or tail of the queue.

4 To set a software interlock realized with a queue, you can use the following:

```
      INSQUE  ...              ; Was queue empty?
      BEQL    1$               ; Yes
      CALL    WAIT(...)        ; No, wait
1$:
```

5 During access validation, any access that cannot be completed results in a memory management exception, even though the queue insertion is not started.

REMQHI

Remove Entry from Queue at Head, Interlocked

FORMAT

opcode header.aq, addr.wl

condition codes

```
if {removal succeeded} then
      begin
      N ←— 0;
      Z ←— (header) EQL 0; ! Queue empty after removal
      V ←— {queue empty before this instruction};
      C ←— 0;
      end;
else
      begin
      N ←— 0;
      Z ←— 0;
      V ←— 1; ! Did not remove anything
      C ←— 1; ! Secondary interlock failed
      end;
```

exceptions

reserved operand

opcodes

5E	REMQHI	Remove Entry from Queue at Head, Interlocked

DESCRIPTION

If the secondary interlock is clear, the queue entry following the header is removed from the queue and the address operand is replaced by the address of the entry removed. If the queue was empty prior to this instruction, or if the secondary interlock failed, the condition code V-bit is set; otherwise it is cleared.

If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (see Appendix E), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes

1 Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

2 The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented so that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3 To release a software interlock realized with a queue, you can use the following:

```
1$:     REMQHI  ...             ; Removed last?
        BEQL    2$              ; Yes
        BCS     1$              ; Try removing again
        CALL    ACTIVATE(...)   ; Activate other waiters
2$:
```

4 To remove entries until the queue is empty, you can use the following:

```
1$:     REMQHI  ...             ; Anything removed?
        BVS     2$              ; No
          .
        process removed entry
          .
        BR      1$              ;
          .
2$      BCS     1$              ; Try removing again
        queue empty
```

5 During access validation, any access that cannot be completed results in a memory management exception, even though the queue removal is not started.

6 A reserved operand fault occurs if **header** or (**header** + (**header**)) is an address that is not quadword aligned (that is, <2:0> NEQU 0) or if (**header**)<2:1> is not zero. A reserved operand fault also occurs if the header address operand equals the address of the **addr** operand. In this case, the queue is not altered.

REMQTI

Remove Entry from Queue at Tail, Interlocked

FORMAT

opcode *header.aq, addr.wl*

condition codes

```
if {removal succeeded} then
      begin
      N ←— 0;
      Z ←— (header + 4) EQL 0; ! Queue empty after removal
      V ←— {queue empty before this instruction};
      C ←— 0;
      end;
else
      begin
      N ←— 0;
      Z ←— 0;
      V ←— 1; ! Did not remove anything
      C ←— 1; ! Secondary interlock failed
      end;
```

exceptions

reserved operand

opcodes

5F	REMQTI	Remove Entry from Queue at Tail, Interlocked

DESCRIPTION

If the secondary interlock is clear, the queue entry preceding the header is removed from the queue and the address operand is replaced by the address of the entry removed. If the queue was empty prior to this instruction, or if the secondary interlock failed, the condition code V-bit is set; otherwise it is cleared.

If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, even in a multiprocessor environment. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (see Appendix E), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes

1 Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

2 The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented to allow cooperating software processes in a multiprocessor system to access a shared list without additional synchronization.

3 To release a software interlock realized with a queue, you can use the following:

```
1$:     REMQTI  ...             ; Removed last?
        BEQL    2$              ; Yes
        BCS     1$              ; Try removing again
        CALL    ACTIVATE(...)   ; Activate other waiters
2$:
```

4 To remove entries until the queue is empty, you can use the following:

```
1$:     REMQTI  ...             ; Anything removed?
        BVS     2$              ; No
          .
        process removed entry
          .
        BR      1$              ;
          .
2$:     BCS     1$              ; Try removing again
        queue empty
```

5 During access validation, any access which cannot be completed results in a memory management exception, even though the queue removal is not started.

6 A reserved operand fault occurs if **header**, (**header** + 4), or (**header** + (**header** + 4)+4) is an address that is not quadword aligned (that is, <2:0> NEQU 0), or if (**header**)<2:1> is not zero. A reserved operand fault also occurs if the header address operand equals the address of the **addr** operand. In this case, the queue is not altered.

REMQUE

Remove Entry from Queue

FORMAT

opcode entry.ab,addr.wl

condition codes

N	⟵ (entry) LSS (entry+4);
Z	⟵ (entry) EQL (entry+4); ! Queue empty
V	⟵ (entry) EQL (entry+4); ! No entry to remove
C	⟵ (entry) LSSU (entry+4);

exceptions

None.

opcodes

0F	REMQUE	Remove Entry from Queue

DESCRIPTION

The queue entry specified by the entry operand is removed from the queue. The address operand is replaced by the address of the entry removed. If there was no entry in the queue to be removed, the condition code V-bit is set; otherwise it is cleared. If the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (see Appendix E), the queue is left in a consistent state.

Notes

1 The following three types of removal can be performed by suitable choice of entry operand:

- Remove at head:

```
REMQUE  @h,addr         ; h is queue header
```

- Remove at tail:

```
REMQUE  @h+4,addr       ; h is queue header
```

- Remove arbitrary entry:

```
REMQUE  entry,addr
```

2 Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

3 The INSQUE and REMQUE instructions are implemented so that cooperating software processes in a single processor may access a shared list without additional synchronization, if the insertions and removals are only at the head or tail of the queue.

4 To release a software interlock realized with a queue, you can use the following:

```
        REMQUE  ...             ; Queue empty?
        BEQL    1$              ; Yes
        CALL    ACTIVATE(...)   ; Activate other waiters
1$:
```

5 To remove entries until the queue is empty, you can use the following:

```
1$:     REMQUE  ...             ; Anything removed?
        BVS     EMPTY           ; No
          .
          .
          .
        BR      1$
```

6 During access validation, any access which cannot be completed results in a memory management exception, even though the queue removal is not started.