9.2.8 Floating-Point Instructions

Floating-point instructions operate on the following four data types:

- F_floating, standard on all VAX processors
- D_floating, standard on all VAX processors
- G_floating, optional on the VAX–11/780 and the VAX–11/750, and standard on the VAX–11/730
- H_floating, optional on the VAX–11/780 and the VAX–11/750, and standard on the VAX–11/730

To be consistent with the floating-point instruction set, which faults on reserved operands (see Chapter 8), software-implemented floating-point functions (for example, the absolute function) should verify that no input operands are reserved. An easy way to do this is a floating move or test of the input operands.

To make high-speed, floating-point operations easier, restrictions are placed on the addressing mode combinations usable within a single floating-point instruction. These combinations involve the logically inconsistent simultaneous use of a value as both a floating-point operand and an address.

If, within the same instruction, you use the contents of register Rn as both a part of a floating-point input operand (an .rf, .rd, .rg, .rh, .mf, .md, .mg, or .mh operand) and as an address in an addressing mode that modifies Rn (autoincrement, autodecrement, or autoincrement deferred), the value of the floating-point operand is UNPREDICTABLE.

9.2.8.1 Introduction

Mathematically, a floating-point number may be defined as having the following form:

$(+or-)(2 * *K) * f$

where **K** is an integer and **f** is a nonnegative fraction. For a nonvanishing number, **K** and **f** are uniquely determined by imposing the following condition:

```
1/2 LEQ f LSS 1.
```

The fractional factor, **f**, of the number is then said to be **binary normalized**. For the number 0, **f** must be assigned the value zero, and the value of **K** is indeterminate.

VAX derives these floating-point data formats from this mathematical representation for floating-point numbers. Four types of floating-point data are provided: the two standard PDP–11 formats (F_floating and D_floating), and two extended-range formats (G_floating and H_floating). Single-precision, or floating, data is 32 bits long. Double-precision, or D_floating, data is 64 bits long. Extended-range double-precision, or G_floating, data is 64 bits long. Extended-range quadruple-precision, or H_floating, data is 128 bits long. Use sign magnitude notation as follows:

1 Nonzero floating-point numbers:

The most significant bit of the floating-point data is the sign bit: 0 for positive and 1 for negative.

The fractional factor **f** is assumed normalized, so that its most significant bit must be 1. This 1 is the "hidden" bit: it is not stored in the data word, but the hardware restores it before carrying out arithmetic operations. The F_floating and D_floating data types use 23 and 55 bits, respectively, for **f**, which, with the hidden bit, imply effective significance of 24 bits and 56 bits for arithmetic operations. The extended-range (G_floating and H_floating) data types use 52 and 112 bits, respectively, for **f**, which, with the hidden bit, imply effective significance of 53 and 113 bits for arithmetic operations.

In the F_floating and D_floating data types, 8 bits are reserved for the storage of the exponent **K** in excess 128 notation. Thus, exponents from –128 to +127 could be represented, in biased form, by 0 to 255. For reasons given later, a biased exponent of zero (the true exponent of –128) is reserved for floating-point zero. Thus, for F_floating and D_floating data types, exponents are restricted to the range –127 to +127 inclusive or, in excess 128 notation, 1 to 255.

In the G_floating data type, 11 bits are reserved for the storage of the exponent in excess 1024 notation. In the H_floating data type, 15 bits are reserved for the storage of the exponent in excess 16,384 notation. A biased exponent of zero is reserved for floating-point zero. Thus, exponents are restricted to –1023 to +1023 inclusive (in excess notation, 1 to 2047), and –16,383 to +16,383 inclusive (in excess notation, 1 to 32,767) for G_floating and H_floating data types, respectively.

2 Floating-point 0:

Because of the hidden bit, the fractional factor is not available to distinguish between zero and nonzero numbers whose fractional factor is exactly 1/2. Therefore, the VAX reserves a sign-exponent field of zero for this purpose. Any positive floating-point number with a biased exponent of zero is treated as if it were an exact zero by the floating-point instruction set. In particular, a floating-point operand whose bits are all zeros is treated as zero, and this is the format generated by all floating-point instructions for which the result is zero.

3 The reserved operands:

A reserved operand is defined to be any bit pattern with a sign bit of 1 and a biased exponent of zero. On the VAX, all floating-point instructions generate a fault if a reserved operand is encountered. A reserved operand is never generated as a result of a floating-point instruction.

9.2.8.2 Overview of the Instruction Set

The VAX has the standard arithmetic operations ADD, SUB, MUL, and DIV implemented for all four floating-point data types. The results of these operations are always rounded, as described in Section 9.2.8.3. In addition, VAX has two composite operations, EMOD and POLY, also implemented for all four floating-point data types. EMOD generates a product of two operands and then separates the product into its integer and fractional terms. POLY evaluates a polynomial, given the degree, the argument, and a pointer to a table of coefficients. Details on the operation of EMOD and POLY are given in their respective descriptions. All of these instructions are subject to the rounding errors associated with floating-point operations, as well as to exponent overflow and underflow. Accuracy is discussed in Section 9.2.8.3. Exceptions are discussed in Appendix E.

The VAX architecture also has a complete set of instructions for conversion from integer arithmetic types (byte, word, longword) to all floating types (F_floating, D_floating, G_floating, H_floating), and vice versa. The VAX architecture also has a set of instructions for conversion between all of the floating types except between D_floating and G_floating. Many of these instructions are exact, in the sense defined in Section 9.2.8.3. However, a few may generate rounding error, floating overflow, or floating underflow, or induce integer overflow. Details are given in the description of the CVT instructions.

The following move-type instructions are always exact: MOV, NEG, CLR, CMP, and TST. The ACB (Add Compare and Branch) instruction is subject to rounding errors, overflow, and underflow.

All of the floating-point instructions on the VAX architecture fault if they encounter a reserved operand. Floating-point instructions also fault on the occurrence of floating overflow or divide by zero, and the condition codes are UNPREDICTABLE. The FU bit in the processor status word (PSW) is available to enable or disable an exception on underflow. If the FU bit is clear, no exception occurs on underflow and zero is returned as the result. If the FU bit is set, a fault occurs on underflow. Further details on the actions taken if any of these exceptions occurs are included in the descriptions of the instructions and discussed in Appendix E.

9.2.8.3 Accuracy

This section discusses general comments on the accuracy of the VAX floating-point instruction set. The descriptions of the individual instructions may include additional details on their accuracy.

An instruction is defined to be exact if its result, extended on the right by an infinite sequence of zeros, is identical to that of an infinite precision calculation involving the same operands. The prior accuracy of the operands is ignored. For all arithmetic operations except DIV, a zero operand implies that the instruction is exact. The instruction is exact for DIV if the 0 operand is the dividend. If the 0 operand is the divisor, division is undefined and the instruction faults.

For nonzero floating-point operands, the fractional factor is binary normalized with 24 or 56 bits for single-precision (F_floating) or double-precision (D_floating), respectively; and 53 or 113 bits for extended-range double-precision (G_floating), and extended-range quadruple-precision (H_floating), respectively. The ADD, SUB, MUL, and DIV instructions require an overflow bit (on the left) and two guard bits (on the right) to guarantee the return of a rounded result identical to the corresponding infinite precision operation rounded to the specified word length. With these two guard bits, a rounded result has an error bound of 1/2 LSB (least significant bit).

Note that an arithmetic result is exact if no nonzero bits are lost in chopping the infinite precision result to the data length to be stored. Chopping is defined to mean that the 24 (F_floating), 56 (D_floating), 53 (G_floating), or 113 (H_floating) high-order bits of the normalized fractional factor of a result are stored; the rest of the bits are discarded. The first bit lost in chopping is referred to as the "rounding" bit. The value of a rounded result is related to the chopped result as follows:

- If the rounding bit is 1, the rounded result is the chopped result incremented by an LSB (least significant bit).
- If the rounding bit is zero, the rounded and chopped results are identical.

All VAX processors implement rounding to produce results identical to the results produced by the following algorithm: add a 1 to the rounding bit and propagate the carry, if it occurs. Note that a renormalization may be required after rounding takes place. If this occurs, the new rounding bit will be 0; therefore, it can occur only once. The following statements summarize the relations among chopped, rounded, and true (infinite precision) results:

- If a stored result is exact:
  - *roundedvalue* = *choppedvalue* = *truevalue*
- If a stored result is not exact:
  - Its magnitude is always less than that of the true result for chopping.
  - Its magnitude is always less than that of the true result for rounding if the rounding bit is zero.
  - Its magnitude is greater than that of the true result for rounding if the rounding bit is 1.

9.2.8.4 Instruction Descriptions

The following instructions are described in this section:

	Description and Opcode	Number of Instructions
1.	Add 2 Operand ADD{F,D,G,H}2 add.rx, sum.mx	4
2.	Add 3 Operand ADD{F,D,G,H}3 add1.rx, add2.rx, sum.wx	4
3.	Clear CLR{L=F,Q=D=G,O=H} dst.wx	3
4.	Compare CMP{F,D,G,H} src1.rx, src2.rx	4
5.	Convert CVT{F,D,G,H}{B,W,L,F,D,G,H} src.rx, dst.wy CVT{B,W,L}{F,D,G,H} src.rx, dst.wy All pairs except FF,DD,GG,HH,DG, and GD	34
6.	Convert Rounded CVTR{F,D,G,H}L src.rx, dst.wl	4
7.	Divide 2 Operand DIV{F,D,G,H}2 divr.rx, quo.mx	4
8.	Divide 3 Operand DIV{F,D,G,H}3 divr.rx, divd.rx, quo.wx	4
9.	Extended Modulus EMOD{F,D} mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx EMOD{G,H} mulr.rx, mulrx.rw, muld.rx, int.wl, fract.wx	4
10.	Move Negated MNEG{F,D,G,H} src.rx, dst.wx	4
11.	Move MOV{F,D,G,H} src.rx, dst.wx	4
12.	Multiply 2 Operand MUL{F,D,G,H}2 mulr.rx, prod.mx	4
13.	Multiply 3 Operand MUL{F,D,G,H}3 mulr.rx, muld.rx, prod.wx	4
14.	Polynomial Evaluation F_floating POLYF arg.rf, degree.rw, tbladdr.ab, {R0-3.wl}	1
15.	Polynomial Evaluation D_floating POLYD arg.rd, degree.rw, tbladdr.ab, {R0-5.wl}	1
16.	Polynomial Evaluation G_floating POLYG arg.rg, degree.rw, tbladdr.ab, {R0-5.wl}	1

	Description and Opcode	Number of Instructions
17.	Polynomial Evaluation H_floating POLYH arg.rh, degree.rw, tbladdr.ab, {R0-5.wl,-16(SP):-1(SP).wb}	1
18.	Subtract 2 Operand SUB{F,D,G,H}2 sub.rx, dif.mx	4
19.	Subtract 3 Operand SUB{F,D,G,H}3 sub.rx, min.rx, dif.wx	4
20.	Test TST{F,D,G,H} src.rx	4

The following floating-point instructions are described in Section 9.2.4.

	Description and Opcode	Number of Instructions
1.	Add Compare and Branch ACB{F,D,G,H} limit.rx, add.rx, index.mx, displ.bw Compare is LE on positive add, GE on negative add.	4

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 ⟵ 0;
C ⟵ 0;

exceptions

floating overflow
floating underflow
reserved operand

opcodes

40	ADDF2	Add F_floating 2 Operand
41	ADDF3	Add F_floating 3 Operand
60	ADDD2	Add D_floating 2 Operand
61	ADDD3	Add D_floating 3 Operand
40FD	ADDG2	Add G_floating 2 Operand
41FD	ADDG3	Add G_floating 3 Operand
60FD	ADDH2	Add H_floating 2 Operand
61FD	ADDH3	Add H_floating 3 Operand

DESCRIPTION

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

Notes

1 On a reserved operand fault, the sum operand is unaffected, and the condition codes are UNPREDICTABLE.

2 On floating underflow, if FU is set, a fault occurs. Zero is stored as the result of floating underflow only if FU is clear. On a floating underflow fault, the sum operand is unaffected. If FU is clear, the sum operand is replaced by zero, and no exception occurs.

3 On floating overflow, the instruction faults, the sum operand is unaffected, and the condition codes are UNPREDICTABLE.

CLR

Clear

FORMAT

opcode dst.wx

condition codes

N ⟵ 0;
Z ⟵ 1;
V ⟵ 0;
C ⟵ C;

exceptions

None.

opcodes

D4	CLRF	Clear F_floating
7C	CLRD	Clear D_floating,
	CLRG	Clear G_floating
7CFD	CLRH	Clear H_floating

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 ⟵ 0;

exceptions

reserved operand

opcodes

51	CMPF	Compare F_floating
71	CMPD	Compare D_floating
51FD	CMPG	Compare G_floating
71FD	CMPH	Compare H_floating

DESCRIPTION

The source 1 operand is compared with the source 2 operand. The only action is to affect 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
floating overflow
floating underflow
reserved operand

opcodes

4C	CVTBF	Convert Byte to F_floating
6C	CVTBD	Convert Byte to D_floating
4CFD	CVTBG	Convert Byte to G_floating
6CFD	CVTBH	Convert Byte to H_floating
4D	CVTWF	Convert Word to F_floating
6D	CVTWD	Convert Word to D_floating
4DFD	CVTWG	Convert Word to G_floating
6DFD	CVTWH	Convert Word to H_floating
4E	CVTLF	Convert Long to F_floating
6E	CVTLD	Convert Long to D_floating
4EFD	CVTLG	Convert Long to G_floating
6EFD	CVTLH	Convert Long to H_floating
48	CVTFB	Convert F_floating to Byte
68	CVTDB	Convert D_floating to Byte
48FD	CVTGB	Convert G_floating to Byte
68FD	CVTHB	Convert H_floating to Byte
49	CVTFW	Convert F_floating to Word
69	CVTDW	Convert D_floating to Word
49FD	CVTGW	Convert G_floating to Word
69FD	CVTHW	Convert H_floating to Word
4A	CVTFL	Convert F_floating to Long

4B	CVTRFL	Convert Rounded F_floating to Long
6A	CVTDL	Convert D_floating to Long
6B	CVTRDL	Convert Rounded D_floating to Long
4AFD	CVTGL	Convert G_floating to Long
4BFD	CVTRGL	Convert Rounded G_floating to Long
6AFD	CVTHL	Convert H_floating to Long
6BFD	CVTRHL	Convert Rounded H_floating to Long
56	CVTFD	Convert F_floating to D_floating
99FD	CVTFG	Convert F_floating to G_floating
98FD	CVTFH	Convert F_floating to H_floating
76	CVTDF	Convert D_floating to F_floating
32FD	CVTDH	Convert D_floating to H_floating
33FD	CVTGF	Convert G_floating to F_floating
56FD	CVTGH	Convert G_floating to H_floating
F6FD	CVTHF	Convert H_floating to F_floating
F7FD	CVTHD	Convert H_floating to D_floating
76FD	CVTHG	Convert H_floating to G_floating

DESCRIPTION

The source operand is converted to the data type of the destination operand, and the destination operand is replaced by the result. The form of the conversion is as follows:

Form	Instructions
Exact	CVTBF, CVTBD, CVTBG, CVTBH, CVTWF, CVTWD, CVTWG, CVTWH, CVTLD, CVTLG, CVTLH, CVTFD, CVTFG, CVTFH, CVTDH, CVTGH
Truncated	CVTFB, CVTDB, CVTGB, CVTHB, CVTFW, CVTDW, CVTGW, CVTHW, CVTFL, CVTDL, CVTGL, CVTHL
Rounded	CVTLF, CVTRFL, CVTRDL, CVTRGL, CVTRHL, CVTDF, CVTGF, CVTHF, CVTHD, CVTHG

Notes

1 Only CVTDF, CVTGF, CVTHF, CVTHD, and CVTHG can result in a floating overflow fault; the destination operand is unaffected, and the condition codes are UNPREDICTABLE.

2 Only converts with a floating-point source operand can result in a reserved operand fault. On a reserved operand fault, the destination operand is unaffected, and the condition codes are UNPREDICTABLE.

3 Only converts with an integer destination operand can result in integer overflow. On integer overflow, the destination operand is replaced by the low-order bits of the true result.

4 Only CVTGF, CVTHF, CVTHD, and CVTHG can result in floating underflow. If FU is set, a fault occurs. On a floating underflow fault, the destination operand is unaffected. If FU is clear, the destination operand is replaced by zero, and no exception occurs.

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 ⟵ 0;
C ⟵ 0;

exceptions

floating overflow
floating underflow
divide by zero
reserved operand

opcodes

46	DIVF2	Divide F_floating 2 Operand
47	DIVF3	Divide F_floating 3 Operand
66	DIVD2	Divide D_floating 2 Operand
67	DIVD3	Divide D_floating 3 Operand
46FD	DIVG2	Divide G_floating 2 Operand
47FD	DIVG3	Divide G_floating 3 Operand
66FD	DIVH2	Divide H_floating 2 Operand
67FD	DIVH3	Divide H_floating 3 Operand

DESCRIPTION

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

Notes

1 On a reserved operand fault, the quotient operand is unaffected, and the condition codes are UNPREDICTABLE.

2 On floating underflow, if FU is set, a fault occurs. On a floating underflow fault, the quotient operand is unaffected. If FU is clear, the quotient operand is replaced by zero, and no exception occurs.

3 On floating overflow, the instruction faults, the quotient operand is unaffected, and the condition codes are UNPREDICTABLE.

4 On divide by zero, the quotient operand, and condition codes are affected as in note 3.

EMOD

Extended Multiply and Integerize

FORMAT

EMODF and EMODD:
opcode mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx
EMODG and EMODH:
opcode mulr.rx, mulrx.rw, muld.rx, int.wl, fract.wx

condition codes

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

exceptions

integer overflow
floating underflow
reserved operand

opcodes

54	EMODF	Extended Multiply and Integerize F_floating
74	EMODD	Extended Multiply and Integerize D_floating
54FD	EMODG	Extended Multiply and Integerize G_floating
74FD	EMODH	Extended Multiply and Integerize H_floating

DESCRIPTION

The multiplier extension operand is concatenated with the multiplier operand to gain 8 (EMODD and EMODF), 11 (EMODG), or 15 (EMODH) additional low-order fraction bits. The low-order 5 or 1 bits of the 16-bit multiplier extension operand are ignored by the EMODG and EMODH instructions, respectively. The multiplicand operand is multiplied by the extended multiplier operand. The multiplication result is equivalent to the exact product truncated (before normalization) to a fraction field of 32 bits in F_floating, 64 bits in D_floating and G_floating, and 128 bits in H_floating. The result is regarded as the sum of an integer and fraction of the same sign. The integer operand is replaced by the integer part of the result, and the fraction operand is replaced by the rounded fractional part of the result.

Notes

1 On a reserved operand fault, the integer operand, and the fraction operand are unaffected. The condition codes are UNPREDICTABLE.

2 On floating underflow, if FU is set, a fault occurs. On a floating underflow fault, the integer and fraction parts are unaffected. If FU is clear, the integer and fraction parts are replaced by zero, and no exception occurs.

3 On integer overflow, the integer operand is replaced by the low-order bits of the true result.

4 Floating overflow is indicated by integer overflow; however, integer overflow is possible in the absence of floating overflow.

5 The signs of the integer and fraction are the same unless integer overflow results.

6 Because the fraction part is rounded after separation of the integer part, it is possible that the value of the fraction operand is 1.

MNEG

Move Negated

FORMAT

opcode src.rx, dst.wx

condition codes

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

exceptions

reserved operand

opcodes

52	MNEGF	Move Negated F_floating
72	MNEGD	Move Negated D_floating
52FD	MNEGG	Move Negated G_floating
72FD	MNEGH	Move Negated H_floating

DESCRIPTION

The destination operand is replaced by the negative of 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

reserved operand

opcodes

50	MOVF	Move F_floating
70	MOVD	Move D_floating
50FD	MOVG	Move G_floating
70FD	MOVH	Move H_floating

DESCRIPTION

The destination operand is replaced by the source operand.

Note

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

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 ⟵ 0;
C ⟵ 0;

exceptions

floating overflow
floating underflow
reserved operand

opcodes

44	MULF2	Multiply F_floating 2 Operand
45	MULF3	Multiply F_floating 3 Operand
64	MULD2	Multiply D_floating 2 Operand
65	MULD3	Multiply D_floating 3 Operand
44FD	MULG2	Multiply G_floating 2 Operand
45FD	MULG3	Multiply G_floating 3 Operand
64FD	MULH2	Multiply H_floating 2 Operand
65FD	MULH3	Multiply H_floating 3 Operand

DESCRIPTION

In 2 operand format, the product operand is multiplied by the multiplier operand, and the product operand is replaced by the rounded result. In 3 operand format, the multiplicand operand is multiplied by the multiplier operand, and the product operand is replaced by the rounded result.

Notes

1 On a reserved operand fault, the product operand is unaffected, and the condition codes are UNPREDICTABLE.

2 On floating underflow, if FU is set, a fault occurs. On a floating underflow fault, the product operand is unaffected. If FU is clear, the product operand is replaced by zero, and no exception occurs.

3 On floating overflow, the instruction faults, the product operand is unaffected, and the condition codes are UNPREDICTABLE.

POLY

Polynomial Evaluation

FORMAT

opcode arg.rx, degree.rw, tbladdr.ab

condition codes

N ⟵ R0 LSS 0;
Z ⟵ R0 EQL 0;
V ⟵ 0;
C ⟵ 0;

exceptions

floating overflow
floating underflow
reserved operand

opcodes

55	POLYF	Polynomial Evaluation F_floating
75	POLYD	Polynomial Evaluation D_floating
55FD	POLYG	Polynomial Evaluation G_floating
75FD	POLYH	Polynomial Evaluation H_floating

DESCRIPTION

The table address operand points to a table of polynomial coefficients. The coefficient of the highest-order term of the polynomial is pointed to by the table address operand. The table is specified with lower-order coefficients stored at increasing addresses. The data type of the coefficients is the same as the data type of the argument operand. The evaluation is carried out by Horner's method, and the contents of R0 (R1′R0 for POLYD and POLYG, R3′R2′R1′R0 for POLYH) are replaced by the result. The result computed is:

```
if d = degree
and x = arg
result = C[0]+x**0 + x*(C[1] + x*(C[2] + ... x*C[d]))
```

The unsigned word degree operand specifies the highest-numbered coefficient to participate in the evaluation. POLYH requires four longwords on the stack to store **arg** in case the instruction is interrupted.

Notes

1 After execution:

POLYF:
R0 = result
R1 = 0
R2 = 0

R3 = table address + degree*4 + 4
POLYD and POLYG:
R0 = high-order part of result
R1 = low-order part of result
R2 = 0
R3 = table address + degree*8 + 8
R4 = 0
R5 = 0
POLYH:
R0 = highest-order part of result
R1 = second-highest-order part of result
R2 = second-lowest-order part of result
R3 = lowest-order part of result
R4 = 0
R5 = table address + degree*16 + 16

2 On a floating fault:

- If PSL<FPD> = 0, the instruction faults, and all relevant side effects are restored to their original state.
- If PSL<FPD> = 1, the instruction is suspended, and the state is saved in the general registers as follows:

```
POLYF:
R0 = tmp3             ! Partial result after iteration
                      !   prior to the one causing the
                      !   overflow/underflow
R1 = arg
R2<7:0> = tmp1        ! Number of iterations remaining
R2<31:8> = implementation specific
R3 = tmp2             ! Points to table entry causing
                      !   exception

POLYD and POLYG:
R1'R0 = tmp3          ! Partial result after iteration
                      !   prior to the one causing the
                      !   overflow/underflow
R2<7:0> = tmp1        ! Number of iterations remaining
R2<31:8> = implementation specific
R3 = tmp2             ! Points to table entry causing
                      !   exception
R5'R4 = arg

POLYH:
R3'R2'R1'R0 = tmp3    ! Partial result after iteration
                      !   prior to the one causing the
                      !   overflow/underflow
R4<7:0> = tmp1        ! Number of iterations remaining
R4<31:8> = implementation specific
R5 = tmp2             ! Points to table entry causing
                      !   exception
```

arg is saved on the stack in use during the faulting instruction.

Implementation-specific information is saved to allow the instruction to continue after possible scaling of the coefficients and partial result by the fault handler.

3 If the unsigned word degree operand is zero and the argument is not a reserved operand, the result is C[0].

4 If the unsigned word degree operand is greater than 31, a reserved operand fault occurs.

5 On a reserved operand fault:

- If PSL<FPD> = 0, the reserved operand is either the degree operand (greater than 31), or the argument operand, or some coefficient.
- If PSL<FPD> = 1, the reserved operand is a coefficient, and R3 (except for POLYH) or R5 (for POLYH) is pointing at the value that caused the exception.
- The state of the saved condition codes and the other registers is UNPREDICTABLE. If the reserved operand is changed and the contents of the condition codes and all registers are preserved, the fault can be continued.

6 On floating underflow after the rounding operation at any iteration of the computation loop, a fault occurs if FU is set. If FU is clear, the temporary result (**tmp3**) is replaced by zero and the operation continues. In this case, the final result may be nonzero if underflow occurred before the last iteration.

7 On floating overflow after the rounding operation at any iteration of the computation loop, the instruction terminates with a fault.

8 If the argument is zero and one of the coefficients in the table is the reserved operand, whether a reserved operand fault occurs is UNPREDICTABLE.

9 For POLYH, some implementations may not save **arg** on the stack until after an interrupt or fault occurs. However, **arg** will always be on the stack if an interrupt or floating fault occurs after FPD is set. If the four longwords on the stack overlap any of the source operands, the results are UNPREDICTABLE.

EXAMPLE

```
; To compute P(x) = C0 + C1*x + C2*x**2
; where C0 = 1.0,  C1 = .5, and C2 = .25

        POLYF   X,#2,PTABLE
        .
        .
        .
PTABLE: .FLOAT  0.25    ; C2
        .FLOAT  0.5     ; C1
        .FLOAT  1.0     ; C0
```

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 ⟵ 0;
C ⟵ 0;

exceptions

floating overflow
floating underflow
reserved operand

opcodes

42	SUBF2	Subtract F_floating 2 Operand
43	SUBF3	Subtract F_floating 3 Operand
62	SUBD2	Subtract D_floating 2 Operand
63	SUBD3	Subtract D_floating 3 Operand
42FD	SUBG2	Subtract G_floating 2 Operand
43FD	SUBG3	Subtract G_floating 3 Operand
62FD	SUBH2	Subtract H_floating 2 Operand
63FD	SUBH3	Subtract H_floating 3 Operand

DESCRIPTION

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

Notes

1 On a reserved operand fault, the difference operand is unaffected, and the condition codes are UNPREDICTABLE.

2 On floating underflow, if FU is set, a fault occurs. Zero is stored as the result of floating underflow only if FU is clear. On a floating underflow fault, the difference operand is unaffected. If FU is clear, the difference operand is replaced by zero, and no exception occurs.

3 On floating overflow, the instruction faults, the difference operand is unaffected, and the condition codes are UNPREDICTABLE.

TST

Test

FORMAT

opcode *src.rx*

condition codes

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

exceptions

reserved operand

opcodes

53	TSTF	Test F_floating
73	TSTD	Test D_floating
53FD	TSTG	Test G_floating
73FD	TSTH	Test H_floating

DESCRIPTION

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

Notes

1 TST*x* **src** is equivalent to CMP*x* **src,** #0, but is 5 (F_floating) or 9 (D_floating or G_floating) or 17 (H_floating) bytes shorter.

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

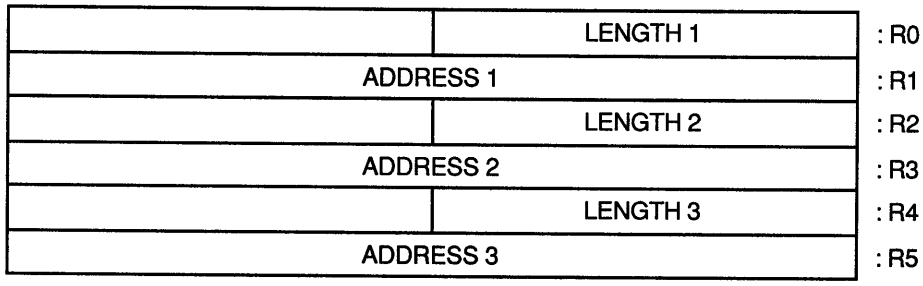
9.2.9 Character String Instructions

A character string is specified by the following two operands:

1 An unsigned word operand that specifies the length of the character string in bytes.

2 The address of the lowest-addressed byte of the character string. This is specified by a byte operand of address access type.

Each of the character string instructions uses general registers R0 to R1, R0 to R3, or R0 to R5 to contain a control block that maintains updated addresses and state during the execution of the instruction. At completion, these registers are available to software to use as string specification operands for a subsequent instruction on a contiguous character string. During the execution of the instructions, pending interrupt conditions are tested. If any conditions are found, the control block is updated, a first-part-done bit is set in the processor status longword (PSL), and the instruction is interrupted (refer to Appendix E). After the interruption, the instruction resumes transparently. The format of the control block is as follows:



ZK-1175A-GE

The fields LENGTH 1, LENGTH 2 (if required), and LENGTH 3 (if required) contain the number of bytes remaining to be processed in the first, second, and third string operands, respectively. The fields ADDRESS 1, ADDRESS 2 (if required), and ADDRESS 3 (if required) contain the address of the next byte to be processed in the first, second, and third string operands, respectively.

Memory access faults do not occur when a zero-length string is specified because no memory reference occurs.

The following instructions are described in this section.

	Description and Opcode	Number of Instructions
1.	Compare Characters 3 Operand CMPC3 len.rw, src1addr.ab, src2addr.ab, {R0-3.wl}	1

	Description and Opcode	Number of Instructions
2.	Compare Characters 5 Operand CMPC5 src1len.rw, src1addr.ab, fill.rb, src2len.rw, src2addr.ab, {R0-3.wl}	1
3.	Locate Character LOCC char.rb, len.rw, addr.ab, {R0-1.wl}	1
4.	Match Characters MATCHC len1.rw, addr1.ab, len2.rw, addr2.ab, {R0-3.wl}	1
5.	Move Character 3 Operand MOVC3 len.rw, srcaddr.ab, dstaddr.ab, {R0-5.wl}	1
6.	Move Character 5 Operand MOVC5 srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab, {R0-5.wl}	1
7.	Move Translated Characters MOVTC srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab, {R0-5.wl}	1
8.	Move Translated Until Character MOVTUC srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw, dstaddr.ab, {R0-5.wl}	1
9.	Scan Characters SCANC len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.wl}	1
10.	Skip Character SKPC char.rb, len.rw, addr.ab, {R0-1.wl}	1
11.	Span Characters SPANC len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.wl}	1

CMPC

Compare Characters

FORMAT

3operand:	*opcode*	*len.rw, src1addr.ab, src2addr.ab*
5operand:	*opcode*	*src1len.rw, src1addr.ab, fill.rb, src2len.rw, src2addr.ab*

condition codes

N ⟵ {first byte} LSS {second byte};
Z ⟵ {first byte} EQL {second byte};
V ⟵ 0;
C ⟵ {first byte} LSSU {second byte};

exceptions

None.

opcodes

29	CMPC3	Compare Characters 3 Operand
2D	CMPC5	Compare Characters 5 Operand

DESCRIPTION

In 3 operand format, the bytes of string1 specified by the length and address1 operands are compared with the bytes of string2 specified by the length and address2 operands. Comparison proceeds until inequality is detected or all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison. In 5 operand format, the bytes of the string1 operand specified by the length1 and address1 operands are compared with the bytes of the string2 operand specified by the length2 and address2 operands. If one string is longer than the other, the shorter string is conceptually extended to the length of the longer by appending (at higher addresses) bytes equal to the fill operand. Comparison proceeds until inequality is detected or all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison. For either CMPC3 or CMPC5, two zero-length strings compare equal (that is, Z is set and N, V, and C are cleared).

Notes

1 After execution of CMPC3:

R0 = Number of bytes remaining in string1 (including byte that terminated comparison); R0 is zero only if strings are equal

R1 = Address of the byte in string1 that terminated comparison; if strings are equal, address of 1 byte beyond string1

R2 = R0

R3 = Address of the byte in string2 that terminated comparison; if strings are equal, address of 1 byte beyond string2

2 After execution of CMPC5:

R0 = Number of bytes remaining in string1 (including byte that terminated comparison); R0 is zero only if string1 and string2 are of equal length and equal or string1 was exhausted before comparison terminated

R1 = Address of the byte in string1 that terminated comparison; if comparison did not terminate before string1 exhausted, address of 1 byte beyond string1

R2 = Number of bytes remaining in string2 (including byte that terminated comparison); R2 is zero only if string2 and string1 are of equal length or string2 was exhausted before comparison terminated

R3 = Address of the byte in string2 that terminated comparison; if comparison did not terminate before string2 was exhausted, address of 1 byte beyond string2

3 If both strings have zero length, condition code Z is set and N, V, and C are cleared just as in the case of two equal strings.

LOCC

Locate Character

FORMAT

opcode char.rb, len.rw, addr.ab

condition codes

N ⟵ 0;
Z ⟵ R0 EQL 0;
V ⟵ 0;
C ⟵ 0;

exceptions

None.

opcodes

3A	LOCC	Locate Character

DESCRIPTION

The character operand is compared with the bytes of the string specified by the length and address operands. Comparison continues until equality is detected or all bytes of the string have been compared. If equality is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.

Notes

1 After execution:

R0 = Number of bytes remaining in the string (including located one) if byte located; otherwise, zero

R1 = Address of the byte located if byte located; otherwise, address of 1 byte beyond the string

2 If the string has zero length, condition code Z is set just as though each byte of the entire string were unequal to character.

MATCHC

Match Characters

FORMAT

opcode objlen.rw, objaddr.ab, srclen.rw, srcaddr.ab

condition codes

N ⟵ 0;
Z ⟵ R0 EQL 0; !match found
V ⟵ 0;
C ⟵ 0;

exceptions

None.

opcodes

39	MATCHC	Match Characters

DESCRIPTION

The source string specified by the source length and source address operands is searched for a substring that matches the object string specified by the object length and object address operands. If the substring is found, the condition code Z-bit is set; otherwise, it is cleared.

Notes

1 After execution:

R0 = If a match occurred, zero; otherwise, the number of bytes in the object string

R1 = If a match occurred, the address of 1 byte beyond the object string; that is, **objaddr** + **objlen**; otherwise, the address of the object string

R2 = If a match occurred, the number of bytes remaining in the source string; otherwise, zero

R3 = If a match occurred, the address of 1 byte beyond the last byte matched; otherwise, the address of 1 byte beyond the source string; that is, **srcaddr** + **srclen**

For zero-length source and object strings, R3 and R1 contain the source and object addresses, respectively.

2 If both strings have zero length, or if the object string has zero length, condition code Z is set, and registers R0 to R3 are left just as though the substring were found.

3 If the source string has zero length and the object string has nonzero length, condition code Z is cleared, and registers R0 to R3 are left just as though the substring were not found.

MOVC

Move Character

FORMAT

3operand:	*opcode*	*len.rw, srcaddr.ab, dstaddr.ab*
5operand:	*opcode*	*srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab*

condition codes

N ⟵ 0; !MOVC3
Z ⟵ 1;
V ⟵ 0;
C ⟵ 0;

N ⟵ srclen LSS dstlen; !MOVC5
Z ⟵ srclen EQL dstlen;
V ⟵ 0;
C ⟵ srclen LSSU dstlen;

exceptions

None.

opcodes

28	MOVC3	Move Character 3 Operand
2C	MOVC5	Move Character 5 Operand

DESCRIPTION

In 3 operand format, the destination string specified by the length and destination address operands is replaced by the source string specified by the length and source address operands. In 5 operand format, the destination string specified by the destination length and destination address operands is replaced by the source string specified by the source length and source address operands. If the destination string is longer than the source string, the highest-addressed bytes of the destination are replaced by the fill operand. If the destination string is shorter than the source string, the highest-addressed bytes of the source string are not moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result.

Notes

1 After execution of MOVC3:

R0 = 0
R1 = Address of 1 byte beyond the source string
R2 = 0
R3 = Address of 1 byte beyond the destination string
R4 = 0
R5 = 0

2 After execution of MOVC5:

R0 = Number of unmoved bytes remaining in source string. R0 is nonzero only if source string is longer than destination string
R1 = Address of 1 byte beyond last byte in source that was moved
R2 = 0
R3 = Address of 1 byte beyond the destination string
R4 = 0
R5 = 0

3 MOVC3 is the preferred way to copy one block of memory to another.

4 MOVC5 with a zero source length operand is the preferred way to fill a block of memory with the fill character.

MOVTC

Move Translated Characters

FORMAT

opcode srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab

condition codes

N ⟵ srclen LSS dstlen;
Z ⟵ srclen EQL dstlen;
V ⟵ 0;
C ⟵ srclen LSSU dstlen;

exceptions

None.

opcodes

2E	MOVTC	Move Translated Characters

DESCRIPTION

The source string specified by the source length and source address operands is translated. It replaces the destination string specified by the destination length and destination address operands. Translation is accomplished by using each byte of the source string as an index into a 256-byte table whose first entry (entry number 0) address is specified by the table address operand. The byte selected replaces the byte of the destination string. If the destination string is longer than the source string, the highest-addressed bytes of the destination string are replaced by the fill operand. If the destination string is shorter than the source string, the highest-addressed bytes of the source string are not translated and moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result.

If the destination string overlaps the translation table, the destination string is UNPREDICTABLE.

Notes

1 After execution:

R0 = Number of untranslated bytes remaining in source string; R0 is nonzero only if source string is longer than destination string

R1 = Address of 1 byte beyond the last byte in source string that was translated

R2 = 0

R3 = Address of the translation table

R4 = 0

R5 = Address of 1 byte beyond the destination string

MOVTUC

Move Translated Until Character

FORMAT

opcode *srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw, dstaddr.ab*

condition codes

N ⟵ srclen LSS dstlen;
Z ⟵ srclen EQL dstlen;
V ⟵ {terminated by escape};
C ⟵ srclen LSSU dstlen;

exceptions

None.

opcodes

2F	MOVTUC	Move Translated Until Character

DESCRIPTION

The source string specified by the source length and source address operands is translated. It replaces the destination string specified by the destination length and destination address operands. Translation is accomplished by using each byte of the source string as an index into a 256-byte table whose first entry address (entry number 0) is specified by the table address operand. The byte selected replaces the byte of the destination string. Translation continues until a translated byte is equal to the escape byte, or until the source string or destination string is exhausted. If translation is terminated because of escape, the condition code V-bit is set; otherwise, it is cleared.

If the destination string overlaps the table, the destination string and registers R0 to R5 are UNPREDICTABLE. If the source and destination strings overlap and their addresses are not identical, the destination string and registers R0 to R5 are UNPREDICTABLE. If the source and destination string addresses are identical, the translation is performed correctly.

Notes

1 After execution:

R0 = Number of bytes remaining in source string (including the byte that caused the escape); R0 is zero only if the entire source string was translated and moved without escape

R1 = Address of the byte that resulted in destination string exhaustion or escape; or if no exhaustion or escape, address of 1 byte beyond the source string

R2 = 0

R3 = Address of the table

R4 = Number of bytes remaining in the destination string

R5 = Address of the byte in the destination string that would have received the translated byte that caused the escape or would have received a translated byte if the source string were not exhausted; or if no exhaustion or escape, the address of 1 byte beyond the destination string

SCANC

Scan Characters

FORMAT

opcode len.rw, addr.ab, tbladdr.ab, mask.rb

condition codes

N ⟵ 0;
Z ⟵ R0 EQL 0;
V ⟵ 0;
C ⟵ 0;

exceptions

None.

opcodes

2A	SCANC	Scan Characters

DESCRIPTION

The assembler successively uses the bytes of the string specified by the length and address operands to index into a 256-byte table whose first entry (entry number 0) address is specified by the table address operand. The logical AND is performed on the byte selected from the table and the mask operand. The operation continues until the result of the AND is nonzero, or until all the bytes of the string have been exhausted. If a nonzero AND result is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.

Notes

1 After execution:

R0 = Number of bytes remaining in the string (including the byte that produced the nonzero AND result); R0 is zero only if there was no nonzero AND result

R1 = Address of the byte that produced the nonzero AND result; if no nonzero result, address of 1 byte beyond the string

R2 = 0

R3 = Address of the table

2 If the string has zero length, condition code Z is set just as though the entire string were scanned.

SKPC

Skip Character

FORMAT

opcode char.rb, len.rw, addr.ab

condition codes

N ⟵ 0;
Z ⟵ R0 EQL 0;
V ⟵ 0;
C ⟵ 0;

exceptions

None.

opcodes

3B	SKPC	Skip Character

DESCRIPTION

The character operand is compared with the bytes of the string specified by the length and address operands. Comparison continues until inequality is detected or all bytes of the string have been compared. If inequality is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.

Notes

1 After execution:

R0 = Number of bytes remaining in the string (including the unequal one) if unequal byte located; otherwise, zero

R1 = Address of the byte located if byte located; otherwise, address of 1 byte beyond the string

2 If the string has zero length, condition code Z is set just as though each byte of the entire string were equal to the character.

SPANC

Span Characters

FORMAT

opcode len.rw, addr.ab, tbladdr.ab, mask.rb

condition codes

N ⟵ 0;
Z ⟵ R0 EQL 0;
V ⟵ 0;
C ⟵ 0;

exceptions

None.

opcodes

2B	SPANC	Span Characters

DESCRIPTION

The assembler successively uses the bytes of the string specified by the length and address operands to index into a 256-byte table whose first entry (entry number 0) address is specified by the table address operand. The logical AND is performed on the byte selected from the table and the mask operand. The operation continues until the result of the AND is zero, or until all the bytes of the string have been exhausted. If a zero AND result is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.

Notes

1 After execution:

R0 = Number of bytes remaining in the string (including the byte that produced the zero AND result); R0 is zero only if there was no zero AND result

R1 = Address of the byte that produced a zero AND result; if no nonzero result, address of 1 byte beyond the string

R2 = 0

R3 = Address of the table

2 If the string has zero length, the condition code Z-bit is set just as though the entire string were spanned.

9.2.10 Cyclic Redundancy Check Instruction

This instruction implements the calculation of a cyclic redundancy check (CRC) string for any CRC polynomial up to 32 bits. Cyclic redundancy checking is an error detection method involving a division of the data stream by a CRC polynomial. The data stream is represented as a standard VAX string in memory. Error detection is accomplished by computing the CRC at the source and again at the destination, comparing the CRC computed at each end. The choice of the polynomial minimizes the number of undetected block errors of specific lengths. The choice of a CRC polynomial is not given here.

The operands of the CRC instruction are a string descriptor, a 16-longword table, and an initial CRC. The string descriptor is a standard VAX operand pair of the length of the string in bytes (up to 65,535) and the starting address of the string. The contents of the table are a function of the CRC polynomial to be used. It can be calculated from the polynomial by the algorithm in the notes. Several common CRC polynomials are also included in the notes. The system uses the initial CRC to start the polynomial correctly. Typically, the CRC has the value zero or –1. If the data stream is represented by a sequence of noncontiguous strings, the value would vary from 0 to -1.

The CRC instruction scans the string and includes each byte of the data stream in the CRC being calculated. The instruction includes the byte of the data stream by performing a logical exclusive OR (XOR) with it and the rightmost 8 bits of the CRC. Then the instruction shifts the CRC right 1 bit and inserts a zero on the left. The instruction uses the rightmost bit of the CRC (lost by the shift) to control the logical XOR operation of the CRC polynomial with the resultant CRC. If the bit is a 1, the instruction performs a logical XOR with the polynomial and the CRC. The instruction again shifts the CRC to the right and performs a conditional logical XOR on the polynomial with the result, for a total of eight times. The actual algorithm used can shift by 1, 2, or 4 bits at a time using the appropriate entries in a specially constructed table. The instruction produces a 32-bit CRC. For shorter polynomials, the result must be extracted from the 32-bit field. The data stream must be either a multiple of 8 bits in length or right-adjusted in the string with leading zero bits.

CRC

Calculate Cyclic Redundancy Check

FORMAT

opcode tbl.ab, inicrc.rl, strlen.rw, stream.ab

condition codes

N ⟵ R0 LSS 0;
Z ⟵ R0 EQL 0;
V ⟵ 0;
C ⟵ 0;

exceptions

None.

opcodes

0B	CRC	Calculate Cyclic Redundancy Check

DESCRIPTION

The CRC of the data stream described by the string descriptor is calculated. The initial CRC is given by **inicrc**; it is normally zero or –1, unless the CRC is calculated in several steps. The result is left in R0. If the polynomial is less than order 32, the result must be extracted from the low-order bits of R0. The CRC polynomial is expressed by the contents of the 16-longword table. See the notes for the calculation of the table.

Notes

1 After execution:

R0 = Result of CRC
R1 = 0
R2 = 0
R3 = Address 1 byte beyond the end of the source string

2 If the data stream is not a multiple of 8 bits, it must be right-adjusted with leading zero fill.

3 If the CRC polynomial is less than order 32, the result must be extracted from the low-order bits of R0.

4 Use the following algorithm to calculate the CRC table given a polynomial expressed:

```
polyn<n> <- {coefficient of x**{order -1-n}}
```

The following routine is system library routine LIB$CRC_TABLE (poly.r1, table.ab). The table is the location of the 64-byte (16-longword) table into which the result will be written.

```
      SUBROUTINE LIB$CRC_TABLE (POLY, TABLE)
      INTEGER*4 POLY, TABLE(0:15), TMP, X
      DO 190 INDEX = 0, 15
      TMP = INDEX
      DO 150 I = 1, 4
      X = TMP .AND. 1
      TMP = ISHFT(TMP,-1)      !logical shift right one bit
      IF (X .EQ. 1) TMP =  TMP .XOR. POLY
150   CONTINUE
      TABLE(INDEX) = TMP
190   CONTINUE
      RETURN
      END
```

5 The following are descriptions of some commonly used CRC polynomials:

```
CRC-16 (used in DDCMP and Bisync)

      polynomial:     x^16 + x^15 + x^2 + 1
      poly:           120001 (octal)
      initialize:     0
      result:         R0<15:0>

CCITT  (used in ADCCP, HDLC, SDLC)

      polynomial:     x^16 + x^12 + x^5 + 1
      poly:           102010 (octal)
      initialize:     -1<15:0>
      result:         one's complement of R0<15:0>

AUTODIN-II

      polynomial:     x^32+x^26+x^23+x^22+x^16+x^12
                        +x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1
      poly:           EDB88320 (hex)
      initialize:     -1<31:0>
      result:         one's complement of R0<31:0>
```

6 The CRC instruction produces an UNPREDICTABLE result unless the table is well-formed, like the one produced in note 3. Note that for any well-formed table, **entry**[0] is always zero and **entry**[8] is always the polynomial expressed as in note 3. The operation can be implemented using shifts of 1, 2, or 4 bits at a time, as follows:

Shift (s)	Steps per Byte (limit)	Table Index	Table Index Multiplier (i)	Use Table Entries
1	8	tmp3<0>	8	[0]=0,[8]
2	4	tmp3<1:0>	4	[0]=0,[4],[8],[12]
4	2	tmp3<3:0>	1	all

7 If the stream has zero length, R0 receives the initial CRC.

9.2.11 Decimal String Instructions

Decimal string instructions operate on packed decimal strings.

The decimal string instructions in this section operate on the following data types:

- Packed decimal string
- Trailing numeric string (overpunched and zoned)
- Leading separate numeric string

Where the phrase "decimal string" is used, it means any of the three data types. Conversion instructions are provided between the data types. Where necessary, a specific data type is identified.

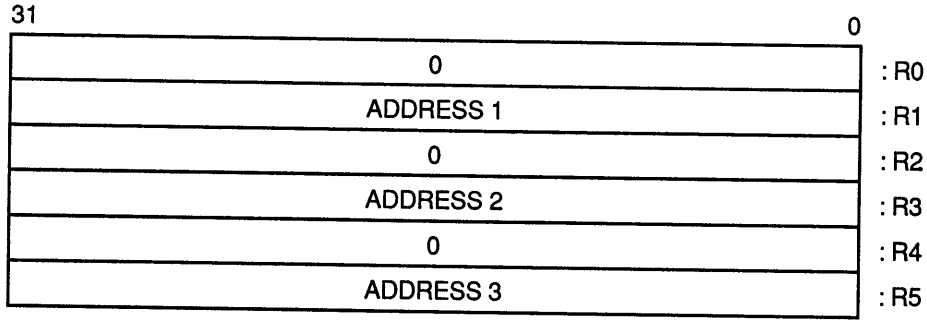
A decimal string is specified by two operands:

1 For all decimal strings, the length is the number of digits in the string. The number of bytes in the string is a function of the length and the type of decimal string referenced (see Chapter 8).

2 The address of the lowest-addressed byte of the string. This byte contains the most significant digit for trailing numeric and packed decimal strings, as well as a sign for leading separate numeric strings. The address is specified by a byte operand of address access type.

Each of the decimal string instructions uses general registers R0 to R3 or R0 to R5 to hold a control block that maintains updated addresses and state during the execution of the instruction. At completion, the registers containing addresses are available to the software for use as string specification operands for a subsequent instruction on the same decimal strings.

During the execution of the instructions, pending interrupt conditions are tested; if any is found, the control block is updated. The first part done is set in the processor status longword (PSL), and the instruction is interrupted (refer to Appendix E). After the interruption, the instruction resumes transparently. The format of the control block at completion is as follows:



ZK-1176A-GE

The fields ADDRESS 1, ADDRESS 2, and ADDRESS 3 (if required) contain the address of the byte containing the most significant digit of the first, second, and third (if required) string operands, respectively.

The decimal string instructions treat decimal strings as integers with the decimal point assumed immediately beyond the least significant digit of the string. If a string in which a result is to be stored is longer than the result, its most significant digits are filled with zeros.

9.2.11.1 Decimal Overflow

Decimal overflow occurs if the destination string is too short to contain all of the digits (excluding leading zeros) of the result. On overflow, the destination string is replaced by the correctly signed least significant digits of the true result (even if the stored result is –0). Note that neither the high nibble of an even-length packed decimal string nor the sign byte of a leading separate numeric string is used to store result digits.

9.2.11.2 Zero Numbers

A zero result has a positive sign for all operations that complete without decimal overflow, except for CVTPT, which does not change a –0 to a +0. However, when digits are lost because of overflow, a zero result receives the sign (positive or negative) of the correct result.

A decimal string with value –0 is treated as identical to a decimal string with value +0. Thus, for example, +0 compares as equal to –0. When condition codes are affected on a –0 result, they are affected as if the result were +0; that is, N is cleared and Z is set.

9.2.11.3 Reserved Operand Exception

A reserved operand abort occurs if the length of a decimal string operand is outside the range 0 to 31, or if an invalid sign or digit is encountered in CVTSP or CVTTP. The program counter (PC) points to the opcode of the instruction causing the exception.

9.2.11.4 UNPREDICTABLE Results

The result of any operation is UNPREDICTABLE if any source decimal string operand contains invalid data. Except for CVTSP and CVTTP, the decimal string instructions do not verify the validity of source operand data.

If the destination operands overlap any source operands, the result of an operation will be UNPREDICTABLE. The destination strings, registers used by the instruction, and condition codes will be UNPREDICTABLE when a reserved operand abort occurs.

9.2.11.5 Packed Decimal Operations

Packed decimal strings generated by the decimal string instructions always have the preferred sign representation: 12 for "+" and 13 for "–". An even-length packed decimal string is always generated with a "0" digit in the high nibble of the first byte of the string.

A packed decimal string contains an invalid nibble if:

- A digit occurs in the sign position
- A sign occurs in a digit position
- A nonzero nibble occurs in the high-order nibble of the lowest-addressed byte in an even length string

9.2.11.6 Zero-Length Decimal Strings

The length of a packed decimal string can be zero. In this case, the value is zero (plus or minus) and 1 byte of storage is occupied. This byte must contain a "0" digit in the high nibble and the sign in the low nibble.

The length of a trailing numeric string can be zero. In this case, no storage is occupied by the string. If a destination operand is a zero-length trailing numeric string, the sign of the operation is lost. Memory access faults do not occur when a zero-length trailing numeric operand is specified because no memory reference occurs. The value of a zero-length trailing numeric string is identically zero.

The length of a leading separate numeric string can be zero. In this case, 1 byte of storage is occupied by the sign. Memory is accessed when a zero-length operand is specified, and a reserved operand abort will occur if an invalid sign is detected. The value of a zero-length leading separate numeric string is zero.

9.2.11.7 Instruction Descriptions

The following instructions are described in this section:

	Description and Opcode	Number of Instructions
1.	Add Packed 4 Operand ADDP4 addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab, {R0-3.wl}	1
2.	Add Packed 6 Operand ADDP6 add1len.rw, add1addr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab, {R0-5.wl}	1
3.	Arithmetic Shift and Round Packed ASHP cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
4.	Compare Packed 3 Operand CMPP3 len.rw, src1addr.ab, src2addr.ab, {R0-3.wl}	1
5.	Compare Packed 4 Operand CMPP4 src1len.rw, src1addr.ab, src2len.rw, src2addr.ab, {R0-3.wl}	1
6.	Convert Long to Packed CVTLP src.rl, dstlen.rw, dstaddr.ab, {R0-3.wl}	1

	Description and Opcode	Number of Instructions
7.	Convert Packed to Long CVTPL srclen.rw, srcaddr.ab, {R0-3.wl}, dst.wl	1
8.	Convert Packed to Leading Separate CVTPS srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
9.	Convert Packed to Trailing CVTPT srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
10.	Convert Leading Separate to Packed CVTSP srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
11.	Convert Trailing to Packed CVTTP srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
12.	Divide Packed DIVP divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quolen.rw, quoaddr.ab, {R0-5.wl, –16(SP):–1(SP).wb}	1
13.	Move Packed MOVP len.rw, srcaddr.ab, dstaddr.ab, {R0-3.wl}	1
14.	Multiply Packed MULP mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab, {R0-5.wl}	1
15.	Subtract Packed 4 Operand SUBP4 sublen.rw, subaddr.ab, diflen.rw, difaddr.ab, {R0-3.wl}	1
16.	Subtract Packed 6 Operand SUBP6 sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, diflen.rw, difaddr.ab, {R0-5.wl}	1

ADDP

Add Packed

FORMAT

opcode	*addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab*
opcode	*add1len.rw, add1addr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab*

condition codes

N ⟵ {sum string} LSS 0;
Z ⟵ {sum string} EQL 0;
V ⟵ {decimal overflow};
C ⟵ 0;

exceptions

reserved operand
decimal overflow

opcodes

20	ADDP4	Add Packed 4 Operand
21	ADDP6	Add Packed 6 Operand

DESCRIPTION

In 4 operand format, the addend string specified by the addend length and addend address operands is added to the sum string specified by the sum length and sum address operands, and the sum string is replaced by the result.

In 6 operand format, the addend1 string specified by the addend1 length and addend1 address operands is added to the addend2 string specified by the addend2 length and addend2 address operands. The sum string specified by the sum length and sum address operands is replaced by the result.

Notes

1 After execution of ADDP4:

R0 = 0
R1 = Address of the byte containing the most significant digit of the addend string
R2 = 0
R3 = Address of the byte containing the most significant digit of the sum string

2 After execution of ADDP6:

R0 = 0

R1 = Address of the byte containing the most significant digit of the addend1 string

R2 = 0

R3 = Address of the byte containing the most significant digit of the addend2 string

R4 = 0

R5 = Address of the byte containing the most significant digit of the sum string

3 The sum string, R0 to R3 (or R0 to R5 for ADDP6) and the condition codes are UNPREDICTABLE if: the sum string overlaps the addend, addend1, or addend2 strings; the addend, addend1, addend2, or sum (4 operand only) strings contain an invalid nibble; or a reserved operand abort occurs.

ASHP

Arithmetic Shift and Round Packed

FORMAT

opcode cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

F8	ASHP	Arithmetic Shift and Round Packed

DESCRIPTION

The source string specified by the source length and source address operands is scaled by a power of 10 specified by the count operand. The destination string specified by the destination length and destination address operands is replaced by the result.

A positive count operand effectively multiplies, a negative count effectively divides, and a zero count just moves and affects condition codes. When a negative count is specified, the result is rounded using the round operand.

Notes

1 After execution:

R0 = 0
R1 = Address of the byte containing the most significant digit of the source string
R2 = 0
R3 = Address of the byte containing the most significant digit of the destination string

2 The destination string, R0 to R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string, the source string contains an invalid nibble, or a reserved operand abort occurs.

3 When the count operand is negative, the result is rounded by decimally adding bits 3:0 of the round operand to the most significant low-order digit discarded and propagating the carry, if any, to higher-order digits. Both the source operand and the round operand are considered to be quantities of the same sign for the purpose of this addition.

4 If bits 7:4 of the round operand are nonzero, or if bits 3:0 of the round operand contain an invalid packed decimal digit, the result is UNPREDICTABLE.

5 When the count operand is zero or positive, the round operand has no effect on the result except as specified in note 4.

6 The round operand is normally 5. Truncation can be accomplished by using a zero round operand.

CMPP

Compare Packed

FORMAT

3operand:	*opcode*	*len.rw, src1addr.ab, src2addr.ab*
4operand:	*opcode*	*src1len.rw, src1addr.ab, src2len.rw, src2addr.ab*

condition codes

N ⟵ {src1 string} LSS {src2 string};
Z ⟵ {src1 string} EQL {src2 string};
V ⟵ 0;
C ⟵ 0;

exceptions

reserved operand

opcodes

35	CMPP3	Compare Packed 3 Operand
37	CMPP4	Compare Packed 4 Operand

DESCRIPTION

In 3 operand format, the source 1 string specified by the length and source 1 address operands is compared to the source 2 string specified by the length and source 2 address operands. The only action is to affect the condition codes.

In 4 operand format, the source 1 string specified by the source 1 length and source 1 address operands is compared to the source 2 string specified by the source 2 length and source 2 address operands. The only action is to affect the condition codes.

Notes

1 After execution of CMPP3 or CMPP4:

R0 = 0
R1 = Address of the byte containing the most significant digit of string1
R2 = 0
R3 = Address of the byte containing the most significant digit of string2

2 R0 to R3 and the condition codes are UNPREDICTABLE if the source strings overlap, if either string contains an invalid nibble, or if a reserved operand abort occurs.

CVTLP

Convert Long to Packed

FORMAT

opcode src.rl, dstlen.rw, dstaddr.ab

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

F9	CVTLP	Convert Long to Packed

DESCRIPTION

The source operand is converted to a packed decimal string. The destination string operand specified by the destination length and destination address operands is replaced by the result.

Notes

1 After execution:

R0 = 0
R1 = 0
R2 = 0
R3 = Address of the byte containing the most significant digit of the destination string

2 The destination string, R0 to R3, and the condition codes are UNPREDICTABLE on a reserved operand abort.

3 Overlapping operands produce correct results.

CVTPL

Convert Packed to Long

FORMAT

opcode srclen.rw, srcaddr.ab, dst.wl

condition codes

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

exceptions

reserved operand
integer overflow

opcodes

36	CVTPL	Convert Packed to Long

DESCRIPTION

The source string specified by the source length and source address operands is converted to a longword, and the destination operand is replaced by the result.

Notes

1 After execution:

R0 = 0
R1 = Address of the byte containing the most significant digit of the source string
R2 = 0
R3 = 0

2 The destination operand, R0 to R3, and the condition codes are UNPREDICTABLE on a reserved operand abort, or if the string contains an invalid nibble.

3 The destination operand is stored after the registers are updated as specified in note 1. You may use R0 to R3 as the destination operand.

4 If the source string has a value outside the range –2,147,483,648 to +2,147,483,647, integer overflow occurs and the destination operand is replaced by the low-order 32 bits of the correctly signed infinite precision conversion. On overflow, the sign of the destination may be different from the sign of the source.

5 Overlapping operands produce correct results.

CVTPS

Convert Packed to Leading Separate Numeric

FORMAT

opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab

condition codes

N ⟵ {src string} LSS 0;
Z ⟵ {src string} EQL 0;
V ⟵ {decimal overflow};
C ⟵ 0;

exceptions

reserved operand
decimal overflow

opcodes

08	CVTPS	Convert Packed to Leading Separate Numeric

DESCRIPTION

The source packed decimal string specified by the source length and source address operands is converted to a leading separate numeric string. The destination string specified by the destination length and destination address operands is replaced by the result.

Conversion is effected by replacing the lowest-addressed byte of the destination string with the ASCII character "+" or "–", determined by the sign of the source string. The remaining bytes of the destination string are replaced by the ASCII representations of the values of the corresponding packed decimal digits of the source string.

Notes

1 After execution:

R0 = 0
R1 = Address of the byte containing the most significant digit of the source string
R2 = 0
R3 = Address of the sign byte of the destination string

2 The destination string, R0 to R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string, the source string contains an invalid nibble, or a reserved operand abort occurs.

3 This instruction produces an ASCII "+" or "–" in the sign byte of the destination string.

4 If decimal overflow occurs, the value stored in the destination might be different from the value indicated by the condition codes (Z and N bits).

5 If the conversion produces a –0 without overflow, the destination leading separate numeric string is changed to a +0 representation.

CVTPT

Convert Packed to Trailing Numeric

FORMAT

opcode srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab

condition codes

N ⟵ {src string} LSS 0;
Z ⟵ {src string} EQL 0;
V ⟵ {decimal overflow};
C ⟵ 0;

exceptions

reserved operand
decimal overflow

opcodes

24	CVTPT	Convert Packed to Trailing Numeric

DESCRIPTION

The source packed decimal string specified by the source length and source address operands is converted to a trailing numeric string. The destination string specified by the destination length and destination address operands is replaced by the result. The condition code N and Z bits are affected by the value of the source packed decimal string.

Conversion is effected by using the highest-addressed byte of the source string (the byte containing the sign and the least significant digit), even if the source string value is –0. The assembler uses this byte as an unsigned index into a 256-byte table whose first entry (entry number 0) address is specified by the table address operand. The byte read from the table replaces the least significant byte of the destination string. The remaining bytes of the destination string are replaced by the ASCII representations of the values of the corresponding packed decimal digits of the source string.

Notes

1 After execution:

R0 = 0
R1 = Address of the byte containing the most significant digit of the source string
R2 = 0
R3 = Address of the most significant digit of the destination string

2 The destination string, R0 to R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string or the table; if the source string or the table contains an invalid nibble; or if a reserved operand abort occurs.

3 The condition codes are computed on the value of the source string even if overflow results. In particular, condition code N is set only if the source is nonzero and contains a minus sign (−).

4 By appropriate specification of the table, you can convert any form of trailing numeric string. See Chapter 8 for the preferred form of trailing overpunch, zoned and unsigned data. In addition, the table can be set up for absolute value, negative absolute value, or negated conversions. The translation table may be referenced even if the length of the destination string is zero.

5 Decimal overflow occurs if the destination string is too short to contain the converted result of a nonzero packed decimal source string (not including leading zeros). Conversion of a source string with zero value never results in overflow; conversion of a nonzero source string to a zero-length destination string results in overflow.

6 If decimal overflow occurs, the value stored in the destination may be different from the value indicated by the condition codes (Z and N bits).

CVTSP

Convert Leading Separate Numeric to Packed

FORMAT

opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

09	CVTSP	Convert Leading Separate Numeric to Packed

DESCRIPTION

The source numeric string specified by the source length and source address operands is converted to a packed decimal string, and the destination string specified by the destination address and destination length operands is replaced by the result.

Notes

1 A reserved operand abort occurs if:

- The length of the source leading separate numeric string is outside the range 0 to 31
- The length of the destination packed decimal string is outside the range 0 to 31
- The source string contains an invalid byte. An invalid byte is any character other than an ASCII "0" to "9" in a digit byte or an ASCII "+", "<space>", or "–" in the sign byte

2 After execution:

R0 = 0
R1 = Address of the sign byte of the source string
R2 = 0
R3 = Address of the byte containing the most significant digit of the destination string

3 The destination string, R0 to R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string, or if a reserved operand abort occurs.

4 **srclen** is the length of the passed string minus the sign byte.

CVTTP

Convert Trailing Numeric to Packed

FORMAT

opcode srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

26	CVTTP	Convert Trailing Numeric to Packed

DESCRIPTION

The source trailing numeric string specified by the source length and source address operands is converted to a packed decimal string, and the destination packed decimal string specified by the destination address and destination length operands is replaced by the result.

Conversion is effected by using the highest-addressed (trailing) byte of the source string as an unsigned index into a 256-byte table whose first entry (entry number 0) is specified by the table address operand. The byte read from the table replaces the highest-addressed byte of the destination string (the byte containing the sign and the least significant digit). The remaining packed digits of the destination string are replaced by the low-order 4 bits of the corresponding bytes in the source string.

Notes

1 A reserved operand abort occurs if:

- The length of the source trailing numeric string is outside the range 0 to 31
- The length of the destination packed decimal string is outside the range 0 to 31
- The source string contains an invalid byte. An invalid byte is any value other than ASCII "0" to "9" in any high-order byte (that is, any byte except the least significant byte)
- The translation of the least significant digit produces an invalid packed decimal digit or sign nibble

2 After execution:

R0 = 0

R1 = Address of the most significant digit of the source string

R2 = 0

R3 = Address of the byte containing the most significant digit of the destination string

3 The destination string, R0 to R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string or the table, or if a reserved operand abort occurs.

4 If the convert instruction produces a –0 without overflow, the destination packed decimal string is changed to a +0 representation, condition code N is cleared, and Z is set.

5 If the length of the source string is zero, the destination packed decimal string is set equal to zero, and the translation table is not referenced.

6 By appropriate specification of the table, you can convert any form of trailing numeric string. See Chapter 8 for the preferred form of trailing overpunch, zoned and unsigned data. In addition, the table can be set up for absolute value, negative absolute value, or negated conversions.

7 If the table translation produces a sign nibble containing any valid sign, the preferred sign representation is stored in the destination packed decimal string.

DIVP

Divide Packed

FORMAT

opcode *divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quolen.rw, quoaddr.ab*

condition codes

N ⟵ {quo string} LSS 0;
Z ⟵ {quo string} EQL 0;
V ⟵ {decimal overflow};
C ⟵ 0;

exceptions

reserved operand
decimal overflow
divide by zero

opcodes

27	DIVP	Divide Packed

DESCRIPTION

The dividend string specified by the dividend length and dividend address operands is divided by the divisor string specified by the divisor length and divisor address operands. The quotient string specified by the quotient length and quotient address operands is replaced by the result.

Notes

1 This instruction allocates a 16-byte workspace on the stack. After execution, the stack pointer (SP) is restored to its original contents, and the contents of {(SP)–16}:{(SP)–1} are UNPREDICTABLE.

2 The division is performed, resulting in the following conditions:

- The absolute value of the remainder (which is lost) is less than the absolute value of the divisor
- The product of the absolute value of the quotient times the absolute value of the divisor is less than or equal to the absolute value of the dividend
- The sign of the quotient is determined by the rules of algebra from the signs of the dividend and the divisor; if the value of the quotient is zero, the sign is always positive

3 After execution:

R0 = 0

R1 = Address of the byte containing the most significant digit of the divisor string

R2 = 0

R3 = Address of the byte containing the most significant digit of the dividend string

R4 = 0

R5 = Address of the byte containing the most significant digit of the quotient string

4 The quotient string, R0 to R5, and the condition codes are UNPREDICTABLE if: the quotient string overlaps the divisor or dividend strings; the divisor or dividend string contains an invalid nibble; the divisor is zero; or a reserved operand abort occurs.

MOVP

Move Packed

FORMAT

opcode len.rw, srcaddr.ab, dstaddr.ab

condition codes

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

exceptions

reserved operand

opcodes

34	MOVP	Move Packed

DESCRIPTION

The destination string specified by the length and destination address operands is replaced by the source string specified by the length and source address operands.

Notes

1 After execution:

R0 = 0
R1 = Address of the byte containing the most significant digit of the source string
R2 = 0
R3 = Address of the byte containing the most significant digit of the destination string

2 The destination string, R0 to R3, and the condition codes are UNPREDICTABLE if: the destination string overlaps the source string; the source string contains an invalid nibble; or a reserved operand abort occurs.

3 If the source is –0, the result is +0, N is cleared, and Z is set.

MULP

Multiply Packed

FORMAT

opcode mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

25	MULP	Multiply Packed

DESCRIPTION

The multiplicand string specified by the multiplicand length and multiplicand address operands is multiplied by the multiplier string specified by the multiplier length and multiplier address operands. The product string specified by the product length and product address operands is replaced by the result.

Notes

1 After execution:

R0 = 0
R1 = Address of the byte containing the most significant digit of the multiplier string
R2 = 0
R3 = Address of the byte containing the most significant digit of the multiplicand string
R4 = 0
R5 = Address of the byte containing the most significant digit of the product string

2 The product string, R0 to R5, and the condition codes are UNPREDICTABLE if: the product string overlaps the multiplier or multiplicand strings; the multiplier or multiplicand strings contain an invalid nibble; or a reserved operand abort occurs.

SUBP

Subtract Packed

FORMAT

4operand:	*opcode*	*sublen.rw, subaddr.ab, diflen.rw, difaddr.ab*
6operand:	*opcode*	*sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, diflen.rw, difaddr.ab*

condition codes

N ⟵ {dif string} LSS 0;
Z ⟵ {dif string} EQL 0;
V ⟵ {decimal overflow};
C ⟵ 0;

exceptions

reserved operand
decimal overflow

opcodes

22	SUBP4	Subtract Packed 4 Operand
23	SUBP6	Subtract Packed 6 Operand

DESCRIPTION

In 4 operand format, the subtrahend string specified by the subtrahend length and subtrahend address operands is subtracted from the difference string specified by the difference length and difference address operands, and the difference string is replaced by the result.

In 6 operand format, the subtrahend string specified by the subtrahend length and subtrahend address operands is subtracted from the minuend string specified by the minuend length and minuend address operands. The difference string specified by the difference length and difference address operands is replaced by the result.

Notes

1 After execution of SUBP4:

R0 = 0
R1 = Address of the byte containing the most significant digit of the subtrahend string
R2 = 0

R3 = Address of the byte containing the most significant digit of the difference string

2 After execution of SUBP6:

R0 = 0

R1 = Address of the byte containing the most significant digit of the subtrahend string

R2 = 0

R3 = Address of the byte containing the most significant digit of the minuend string

R4 = 0

R5 = Address of the byte containing the most significant digit of the difference string

3 The difference string, R0 to R3 (R0 to R5 for SUBP6), and the condition codes are UNPREDICTABLE if: the difference string overlaps the subtrahend or minuend strings; the subtrahend, minuend, or difference (4 operand only) strings contain an invalid nibble; or a reserved operand abort occurs.

9.2.12 The EDITPC Instruction and Its Pattern Operators

The EDITPC instruction implements the common editing functions that occur when handling fixed-format output. The operation consists of converting an input packed decimal number to an output character string and generating characters for the output. When converting digits, options include filling in leading zeros, protecting leading zeros, insertion of floating sign, insertion of floating currency symbol, insertion of special sign representations, and blanking an entire field when it is zero. An example of this operation is a MOVE to a numeric edited (PICTURE) item in COBOL or PL/I. Many other applications are possible.

The operands to the EDITPC instruction are as follows:

1 A **packed decimal string descriptor** (as input). This is a standard VAX operand pair consisting of the length of the decimal string in digits (up to 31) and the starting address of the string.

2 A **pattern** specification, consisting of the starting address of a pattern operation editing sequence. VAX MACRO interprets a pattern specification in the same way as it interprets normal instructions.

3 The **starting address of the output string**. The output string is described by its starting address only, because the pattern defines the length unambiguously.

The EDITPC instruction manipulates two character registers and the four condition codes:

The **fill register** (R2<7:0>) contains the fill character. This is normally an ASCII blank but could be changed to an asterisk (*), for instance, for check protection.

The **sign register** (R2<15:8>) contains the sign character. Initially this register contains either an ASCII blank or a minus sign (–), depending upon the sign of the input. You can change the contents of this register to allow other sign representations such as plus/minus or plus/blank. You can also manipulate it to output special notations such as CR or DB. To implement a floating currency sign, you can change the sign register to the currency sign.

After execution, the condition codes describe the following:

N	The sign of the input
Z	The presence of a zero source
V	An overflow condition
C	The presence of significant digits

Condition code N is determined at the start of the instruction and remains unchanged (except for correcting a –0 input). The processor computes and updates the other condition codes as the instruction proceeds.

When the EDITPC instruction completes processing, registers R0 to R5 contain the values they would normally have after a decimal instruction.

EDITPC

Edit Packed to Character String

FORMAT

opcode srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab

condition codes

N ⟵ {src string} LSS 0; !N <- 0 if src is -0
Z ⟵ {src string} EQL 0;
V ⟵ {decimal overflow}; !nonzero digits lost
C ⟵ {significance};

exceptions

reserved operand
decimal overflow

opcodes

38	EDITPC	Edit Packed to Character String

DESCRIPTION

The destination string specified by the pattern and destination address operands is replaced by the edited version of the source string specified by the source length and source address operands. The editing is performed according to the pattern string, starting at the address of the pattern operand and extending until a pattern end pattern operator (EO$END) is encountered.

The pattern string consists of 1-byte pattern operators. Some pattern operators take no operands. Some take a repeat count that is contained in the rightmost nibble of the pattern operator itself. The rest take a 1-byte operand that immediately follows the pattern operator. This operand is either an unsigned integer length or a byte character.

Table 9–1 lists the pattern operators that can be used with the EDITPC instruction to form a pattern. Subsequent pages define each pattern operator in a format similar to that of the normal instruction descriptions. In each case, if there is an operand, it is either a repeat count (r) from 1 to 15, an unsigned byte length (len), or a character byte (ch). The encoding of the pattern operators is represented graphically in Table 9–2.

See Appendix E for information about exceptions that affect the EDITPC instruction.

Notes

1 A reserved operand abort occurs if **srclen** GTRU 31.

2 The destination string is UNPREDICTABLE if any of the following is true:

- The source string contains an invalid nibble.

- The EO$ADJUST_INPUT operand is outside the range 1 to 31.
- The source and destination strings overlap.
- The pattern and destination strings overlap.

3 After execution, the following general registers have contents as specified:

R0 = Length of source string

R1 = Address of the byte containing the most significant digit of the source string

R2 = 0

R3 = Address of the byte containing the EO$END pattern operator

R4 = 0

R5 = Address of 1 byte beyond the last byte of the destination string

If the destination string is UNPREDICTABLE, R0 to R5 and the condition codes are UNPREDICTABLE.

4 If V is set at the end and DV is enabled, a numeric overflow trap occurs unless the conditions in note 9 are satisfied.

5 The destination length is specified exactly by the pattern operators in the pattern string. If the pattern is incorrectly formed or if it is modified during the execution of the instruction, the length of the destination string is UNPREDICTABLE.

6 If the source is –0, the result may be –0 unless a fixup pattern operator is included (EO$BLANK_ZERO or EO$REPLACE_SIGN).

7 The contents of the destination string and the memory preceding it are UNPREDICTABLE if the length covered by EO$BLANK_ZERO or EO$REPLACE_SIGN is zero, or if it is outside the destination string.

8 If more input digits are requested by the pattern than are specified, a reserved operand abort is taken with R0 = –1 and R3 = location of the pattern operator that requested the extra digit. The condition codes and other registers are as specified in note 11. This abort can not be continued.

9 If fewer input digits are requested by the pattern than are specified, a reserved operand abort is taken with R3 = location of EO$END pattern operator. The condition codes and other registers are as specified in note 11. This abort can not be continued.

10 On an unimplemented or reserved pattern operator, a reserved operand fault is taken with R3 = location of the faulting pattern operator. The condition codes and other registers are as specified in note 11. This fault can be continued as long as the defined register state is manipulated according to the pattern operator description and the state specified as "implementation dependent" is preserved.

11 On a reserved operand exception, as specified in notes 8 to 10, FPD is set and the condition codes and registers are as follows:

N =	{**src** has minus sign}
Z =	All source digits zero so far
V =	Nonzero digits lost
C =	Significance
R0<31:16> =	–(count of source zeros to supply)
R0<15:0> =	Remaining **srclen**
R1 =	Current source location
R2<31:16> =	Implementation dependent
R2<15:8> =	Current contents of sign register
R2<7:0> =	Current contents of fill register
R3 =	Location of edit pattern operator causing exception
R4 =	Implementation dependent
R5 =	Location of next destination byte

Table 9–1 Summary of EDITPC Pattern Operators

Name	Operand	Summary
Insert operators		
EO$INSERT	ch	Insert character, fill if insignificant
EO$STORE_SIGN	—	Insert sign
EO$FILL	r	Insert fill
Move operators		
EO$MOVE	r	Move digits, fill if insignificant
EO$FLOAT	r	Move digits, floating sign
EO$END_FLOAT	—	End floating sign
Fixup operators		
EO$BLANK_ZERO	len	Fill backward when 0
EO$REPLACE_SIGN	len	Replace with fill if –0
Load operators		
EO$LOAD_FILL	ch	Load fill character

Key:

ch—One character
r—Repeat count in the range 1 to 15
len—Length in the range 1 to 255

(continued on next page)

Table 9–1 (Cont.) Summary of EDITPC Pattern Operators

Name	Operand	Summary
Load operators		
EO$LOAD_SIGN	ch	Load sign character
EO$LOAD_PLUS	ch	Load sign character if positive
EO$LOAD_MINUS	ch	Load sign character if negative
Control operators		
EO$SET_SIGNIF	—	Set significance flag
EO$CLEAR_SIGNIF	—	Clear significance flag
EO$ADJUST_INPUT	len	Adjust source length
EO$END	—	End edit

Key:

ch—One character
r—Repeat count in the range 1 to 15
len—Length in the range 1 to 255

Table 9–2 EDITPC Pattern Operator Encoding

Hex	Symbol	Notes
00	EO$END	—
01	EO$END_FLOAT	—
02	EO$CLEAR_SIGNIF	—
03	EO$SET_SIGNIF	—
04	EO$STORE_SIGN	—
05 . . . 1F	—	Reserved to Digital
20 . . . 3F	—	Reserved for all time
40	EO$LOAD_FILL	Character is in next byte
41	EO$LOAD_SIGN	Character is in next byte
42	EO$LOAD_PLUS	Character is in next byte
43	EO$LOAD_MINUS	Character is in next byte
44	EO$INSERT	Character is in next byte
45	EO$BLANK_ZERO	Unsigned length is in next byte
46	EO$REPLACE_SIGN	Unsigned length is in next byte
47	EO$ADJUST_INPUT	Unsigned length is in next byte
48 . . . 5F	—	Reserved to Digital
60 . . . 7F	—	Reserved to CSS and customers
80,90,A0	—	Reserved to Digital

(continued on next page)

Table 9–2 (Cont.) EDITPC Pattern Operator Encoding

Hex	Symbol	Notes
81 . . . 8F	EO$FILL	—
91 . . . 9F	EO$MOVE	Repeat count is <3:0>
A1 . . . AF	EO$FLOAT	—
B0 . . . FE	—	Reserved to Digital
FF	—	Reserved for all time

EO$ADJUST_INPUT

Adjust Input Length

FORMAT

opcode pattern len

pattern operators

47	EO$ADJUST_INPUT	Adjust Input Length

DESCRIPTION

The EO$ADJUST_INPUT pattern operator is followed by an unsigned byte integer length in the range 1 to 31. If the source string has more digits than this length, the excess leading digits are read and discarded. If any discarded digits are nonzero, the overflow is set, significance is set, and zero is cleared. If the source string has fewer digits than this length, a counter is set of the number of leading zeros to supply. This counter is stored as a negative number in R0<31:16>.

Note

If the length is not in the range 1 to 31, the destination string, condition codes, and R0 to R5 are UNPREDICTABLE.

EO$BLANK_ZERO

Blank Backwards when Zero

FORMAT

opcode pattern len

pattern operators

45	EO$BLANK_ZERO	Blank Backwards when Zero

DESCRIPTION

The EO$BLANK_ZERO pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero, then the contents of the fill register are stored into the last length bytes of the destination string.

Notes

1 The length must be nonzero and within the destination string already produced. If it is not, the contents of the destination string and the memory preceding it are UNPREDICTABLE.

2 Use this pattern operator to blank out any characters stored in the destination under a forced significance such as a sign or the digits following the radix point.

EO$END

End Edit

FORMAT

opcode *pattern*

pattern operators

00	EO$END	End Edit

DESCRIPTION

The EO$END pattern operator terminates the edit operation.

Notes

1 If there are still input digits, a reserved operand abort is taken.

2 If the source value is –0, the N condition code is cleared.

EO$END_FLOAT

End Floating Sign

FORMAT

opcode *pattern*

pattern operators

01	EO$END_FLOAT	End Floating Sign

DESCRIPTION

The EO$END_FLOAT pattern operator terminates a floating sign operation. If the floating sign has not yet been placed in the destination (if significance is not set), the contents of the sign register are stored in the destination, and significance is set.

Note

Use this pattern operator after a sequence of one or more EO$FLOAT pattern operators that start with significance clear. The EO$FLOAT sequence can include intermixed EO$INSERTs and EO$FILLs.

EO$FILL

Store Fill

FORMAT

opcode pattern r

pattern operators

8x	EO$FILL	Store Fill

DESCRIPTION

The rightmost nibble of the pattern operator is the repeat count. The EO$FILL pattern operator places the contents of the fill register into the destination the number of times specified by the repeat count.

Note

Use this pattern operator for fill (blank) insertion.

EO$FLOAT

Float Sign

FORMAT

opcode pattern r

pattern operators

Ax	EO$FLOAT	Float Sign

DESCRIPTION

The EO$FLOAT pattern operator moves digits, floating the sign across insignificant digits. The rightmost nibble of the pattern operator is the repeat count. For the number of times specified in the repeat count, the following algorithm is executed:

The next digit from the source is examined. If it is nonzero and significance is not yet set, then the contents of the sign register are stored in the destination, significance is set, and zero is cleared. If the digit is significant, it is stored in the destination; otherwise, the contents of the fill register are stored in the destination.

Notes

1 If r is greater than the number of digits remaining in the source string, a reserved operand abort is taken.

2 Use this pattern operator to move digits with a floating arithmetic sign. The sign must already be set up as for EO$STORE_SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.

3 Use this pattern operator to move digits with a floating currency sign. The sign must already be set up with an EO$LOAD_SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.

EO$INSERT

Insert Character

FORMAT

opcode pattern ch

pattern operators

44	EO$INSERT	Insert Character

DESCRIPTION

The EO$INSERT pattern operator is followed by a character. If significance is set, the character is placed into the destination. If significance is not set, the contents of the fill register are placed into the destination.

Note

Use this pattern operator for inserts that can be made blank (for example, comma (,)) and fixed inserts (for example, slash (/)). Fixed inserts require that significance be set (by EO$SET_SIGNIF or EO$END_FLOAT).

EO$LOAD_

Load Register

FORMAT

opcode pattern ch

pattern operators

40	EO$LOAD_FILL	Load Fill Register
41	EO$LOAD_SIGN	Load Sign Register
42	EO$LOAD_PLUS	Load Sign Register If Plus
43	EO$LOAD_MINUS	Load Sign Register If Minus

DESCRIPTION

The pattern operator is followed by a character. For EO$LOAD_FILL, this character is placed into the fill register. For EO$LOAD_SIGN, this character is placed into the sign register. For EO$LOAD_PLUS, this character is placed into the sign register if the source string has a positive sign. For EO$LOAD_MINUS, this character is placed into the sign register if the source string has a negative sign.

Notes

1 Use EO$LOAD_FILL to set up check protection (* instead of space).

2 Use EO$LOAD_SIGN to set up a floating currency sign.

3 Use EO$LOAD_PLUS to set up a nonblank plus sign.

4 Use EO$LOAD_MINUS to set up a nonminus minus sign (such as CR, DB, or the PL/I +).

EO$MOVE

Move Digits

FORMAT

opcode pattern r

pattern operators

9x	EO$MOVE	Move Digits

DESCRIPTION

The EO$MOVE pattern operator moves digits, filling for insignificant digits. The rightmost nibble of the pattern operator is the repeat count. For the number of times specified in the repeat count, the following algorithm is executed:

The next digit is moved from the source to the destination. If the digit is nonzero, significance is set and zero is cleared. If the digit is not significant (that is, a leading zero), it is replaced by the contents of the fill register in the destination.

Notes

1 If *r* is greater than the number of digits remaining in the source string, a reserved operand abort is taken.

2 Use this pattern operator to move digits without a floating sign. If leading-zero suppression is desired, significance must be clear. If leading zeros should be explicit, significance must be set. A string of EO$MOVEs intermixed with EO$INSERTs and EO$FILLs will handle suppression correctly.

3 If check protection (*) is desired, EO$LOAD_FILL must precede the EO$MOVE.

EO$REPLACE_SIGN

Replace Sign when Zero

FORMAT

opcode pattern len

pattern operators

46	EO$REPLACE_SIGN	Replace Sign when Zero

DESCRIPTION

The EO$REPLACE_SIGN pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero (that is, if Z is set), the contents of the fill register are stored in the byte of the destination string that is **len** bytes before the current position.

Notes

1 The length must be nonzero and within the destination string already produced. If it is not, the contents of the destination string and the memory preceding it are UNPREDICTABLE.

2 You can use this pattern operator to correct a stored sign (EO$END_FLOAT or EO$STORE_SIGN) if a minus was stored and the source value turned out to be zero.

EO$_SIGNIF

Significance

FORMAT

opcode pattern

pattern operators

02	EO$CLEAR_SIGNIF	Clear Significance
03	EO$SET_SIGNIF	Set Significance

DESCRIPTION

The significance indicator is set or cleared. This controls the treatment of leading zeros (leading zeros are zero digits for which the significance indicator is clear).

Notes

1 Use EO$CLEAR_SIGNIF to initialize leading-zero suppression (EO$MOVE) or floating sign (EO$FLOAT) following a fixed insert (EO$INSERT with significance set).

2 Use EO$SET_SIGNIF to avoid leading-zero suppression (before EO$MOVE) or to force a fixed insert (before EO$INSERT).

EO$STORE_SIGN

Store Sign

FORMAT

opcode *pattern*

pattern operators

04	EO$STORE_SIGN	Store Sign

DESCRIPTION

The EO$STORE_SIGN pattern operator places contents of the sign register into the destination.

Note

Use this pattern operator for any nonfloating arithmetic sign. Precede it with either a EO$LOAD_PLUS or EO$LOAD_MINUS, or both, if the default sign convention is not desired.

9.2.13 Other VAX Instructions

The following table lists other VAX instructions:

	Description and Opcode	Number of Instructions
1.	Probe {Read, Write} Accessibility PROBE{R,W} mode.rb, len.rw, base.ab	2
2.	Change Mode CHM{K,E,S,U} param.rw, {-(ySP).w*} Where y=MINU(x, PSL<current_mode>)	4
3.	Return from Exception or Interrupt REI {(SP)+.r*}	1
4.	Load Process Context LDPCTX {PCB.r*, -(KSP).w*}	1
5.	Save Process Context SVPCTX {(SP)+.r*, PCB.w*}	1
6.	Move to Process Register MTPR src.rl, procreg.rl	1
7.	Move from Processor Register MFPR procreg.rl, dst.wl	1
8.	Bugcheck with {word, longword} message identifier BUG{W,L} message.bx	2

PROBE*x*

Probe Accessibility

FORMAT

opcode mode.rb, len.rw, base.ab

condition codes

N ⟵ 0;
Z ⟵ if {both accessible} then 0 else 1;
V ⟵ 0;
C ⟵ C;

exceptions

translation not valid

opcodes

0C	PROBER	Probe Read Accessibility
0D	PROBEW	Probe Write Accessibility

DESCRIPTION

The PROBE instruction checks the read or write accessibility of the first and last byte specified by the base address and the zero-extended length. Note that the bytes in between are not checked. System software must check all pages if they will be accessed between the two end bytes.

The protection is checked against the larger (and therefore less privileged) of the modes specified in bits <1:0> of the mode operand and the previous mode field of the processor status longword (PSL). Note that probing with a mode operand of zero is equivalent to probing the mode specified in the previous-mode field of the PSL.

EXAMPLE

```
MOVL    4(AP),R0        ; Copy the address of first arg so
                        ;   that it cannot be changed
PROBER  #0,#4,(R0)      ; Verify that the longword pointed to
                        ;   by the first arg could be read by
                        ;   the previous access mode
                        ; Note that the arg list itself must
                        ;   already have been probed
BEQL    violation       ; Branch if either byte gives an
                        ;   access violation
MOVQ    8(AP),R0        ; Copy length and address of buffer
                        ;   arg so that they cannot change
PROBER  #0,R0,(R1)      ; Verify that the buffer described by
                        ;   the 2nd and 3rd args could be
                        ;   written by the previous access
                        ;   mode
                        ; Note that the arg list must already
                        ;   have been probed and that the 2nd
                        ;   arg must be known to be less than
                        ;   512
BEQL    violation       ; Branch if either byte gives an
                        ;   access violation
```

Note that for the PROBE instruction, probing an address returns only the accessibility of the pages and has no effect on their residency. However, probing a process address may cause a page fault in the system address space on the per-process page tables.

Notes

1 If the valid bit of the examined page table entry is set, it is UNPREDICTABLE whether the modify bit of the examined page table entry is set by a PROBER. If the valid bit is clear, the modify bit is not changed.

2 Except for note 1, above, the valid bit of the page table entry, PTE<31>, mapping the probed address is ignored.

3 A length violation gives a status of "not-accessible."

4 On the probe of a process virtual address, if the valid bit of the system page table entry is zero, a Translation Not Valid Fault occurs. This allows for the demand paging of the process page tables.

5 On the probe of a process virtual address, if the protection field of the system page table entry indicates No Access, a status of "not-accessible" is given. Thus, a single No Access page table entry in the system map is equivalent to 128 No Access page table entries in the process map.

CHM

Change Mode

FORMAT

opcode *code.rw*

condition codes

N ← 0;
Z ← 0;
V ← 0;
C ← 0;

exceptions

halt

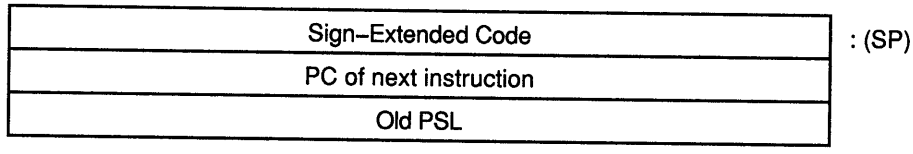
opcodes

BC	CHMK	Change Mode to Kernel
BD	CHME	Change Mode to Executive
BE	CHMS	Change Mode to Supervisor
BF	CHMU	Change Mode to User

DESCRIPTION

Change mode instructions allow processes to change their access mode in a controlled manner. The instruction increases privilege only (decreases the access mode).

A change in mode also results in a change of stack pointers; the old pointer is saved, and the new pointer is loaded. The processor status longword (PSL), program counter (PC), and code passed by the instruction are pushed onto the stack of the new mode. The saved PC addresses the instruction following the CHM*x* instruction. The code is sign extended. After execution, the appearance of the new stack is as follows:



ZK–1177A–GE

The destination mode selected by the opcode is used to obtain a location from the system control block (SCB). This location addresses the CHM*x* dispatcher for the specified mode. If the vector<1:0> code is NEQU 0, then the operation is UNDEFINED.

Notes

1 As usual for faults, any Access Violation or Translation Not Valid fault saves the PC and the PSL, and leaves the stack pointer (SP) as it was at the beginning of the instruction except for any pushes onto the kernel stack.

2 The noninterrupt stack pointers may be fetched and stored either in privileged registers or in their allocated slots in the process control block (PCB). Only LDPCTX and SVPCTX always fetch and store in the PCB. MFPR and MTPR always fetch and store the pointers whether in registers or the PCB.

3 By software convention, negative codes are reserved to CSS and customers.

EXAMPLES

```
CHMK    #7              ; Request the kernel mode service
                        ;   specified by code 7

CHME    #4              ; Request the executive mode service
                        ;   specified by code 4

CHMS    #-2             ; Request the supervisor mode service
                        ;   specified by customer code -2
```

REI

Return from Exception or Interrupt

FORMAT

opcode

condition codes

N ⟵ saved PSL<3>;
Z ⟵ saved PSL<2>;
V ⟵ saved PSL<1>;
C ⟵ saved PSL<0>;

exceptions

reserved operand

opcodes

02	REI	Return from Exception or Interrupt

DESCRIPTION

A longword is popped from the current stack and held in a temporary program counter (PC). A second longword is popped from the current stack and held in a temporary processor status longword (PSL). Validity of the popped PSL is checked. The current stack pointer (SP) is saved, and a new SP is selected according to the new PSL CUR_MOD and IS fields. The level of the highest privilege asynchronous system trap (AST) is checked against the current mode to see whether a pending AST can be delivered. Execution resumes with the instruction being executed at the time of the exception or interrupt. Any instruction latched in the processor is reinitialized.

Notes

1 The exception or interrupt service routine is responsible for restoring any registers saved and for removing any parameters from the stack.

2 As usual for faults, any Access Violation or Translation Not Valid conditions on the stack pops restore the stack pointer and fault.

3 The noninterrupt stack pointers may be fetched and stored either in privileged registers or in their allocated slots in the process control block (PCB). Only LDPCTX and SVPCTX always fetch and store in the PCB. MFPR and MTPR always fetch and store the pointers, whether in registers or in the PCB.

LDPCTX

Load Process Context

FORMAT

opcode

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

reserved operand
privileged instruction

opcodes

06	LDPCTX	Load Process Context

DESCRIPTION

The process control block (PCB) is specified by the privileged register PCB base. The general registers are loaded from the PCB. The memory management registers describing the process address space are also loaded and the process entries in the translation buffer are cleared. Execution is switched to the kernel stack. The program counter (PC) and processor status longword (PSL) are moved from the PCB to the stack, suitable for use by a subsequent REI instruction.

Notes

1 Some processors keep a copy of each of the per-process stack pointers (SPs) in internal registers. In those processors, LDPCTX loads the internal registers from the PCB. Processors that do not keep a copy of all four per-process stack pointers in internal registers keep only the current access mode register in an internal register and switch this with the PCB contents whenever the current access mode field changes.

2 Some implementations may not perform some or all of the reserved operand checks.

SVPCTX

Save Process Context

FORMAT

opcode

condition codes

N ← N;
Z ← Z;
V ← V;
C ← C;

exceptions

privileged instruction

opcodes

07	SVPCTX	Save Process Context

DESCRIPTION

The process control block (PCB) is specified by the privileged register Process Control Block Base. The general registers are saved into the PCB. The program counter (PC) and processor status longword (PSL) currently on the top of the current stack are popped and stored in the PCB. If a SVPCTX instruction is executed when the interrupt stack (IS) is clear, then IS is set, the interrupt stack pointer (ISP) is activated, and interrupt priority level (IPL) is maximized with 1 because of the switch to the IS.

Notes

1 The map, ASTLVL, and PME contents of the process control block (PCB) are not saved because they are rarely changed. Thus, not writing them saves overhead.

2 Some processors keep a copy of each of the per-process stack pointers in internal registers. In those processors, SVPCTX stores the internal registers in the PCB. Processors that do not keep a copy of all four per-process stack pointers in internal registers keep only the current access mode register in an internal register and switch this access mode register with the PCB contents whenever the current access mode field changes.

3 Between the SVPCTX instruction that saves the state for one process and the LDPCTX that loads the state of another, the ISPs may not be referenced by MFPR or MTPR instructions. This implies that interrupt service routines invoked at a priority higher than the lowest one used for context switching must not reference the process stack pointers (SPs).

MTPR

Move to Processor Register

FORMAT

opcode src.rl, procreg.rl

condition codes

```
N   ←— src LSS 0;     ! If register is replaced
Z   ←— src EQL 0;
V   ←— 0;             ! Except TBCHK register
                      ! Please refer to
                      ! Appendix E.

C   ←— C;

N   ←— N;             ! If register is not replaced
Z   ←— Z;
V   ←— V;
C   ←— C;
```

exceptions

reserved operand fault
reserved instruction fault

opcodes

DA	MTPR	Move to Processor Register

DESCRIPTION

Loads the source operand specified by **src** into the processor register specified by **procreg**. The **procreg** operand is a longword that contains the processor register number. Execution may have register-specific side effects.

Notes

1 If the processor internal register does not exist, a reserved operand fault occurs.

2 A reserved instruction fault occurs if instruction execution is attempted in other than kernel mode.

3 A reserved operand fault occurs on a move to a read-only register.

MFPR

Move from Processor Register

FORMAT

opcode procreg.rl, dst.wl

condition codes

```
N   ←— dst LSS 0;        ! If destination is replaced
Z   ←— dst EQL 0;
V   ←— 0;
C   ←— C;

N   ←— N;                ! If destination is not replaced
Z   ←— Z;
V   ←— V;
C   ←— C;
```

exceptions

reserved operand fault
reserved instruction fault

opcodes

DB	MFPR	Move from Processor Register

DESCRIPTION

The destination operand is replaced by the contents of the processor register specified by **procreg**. The **procreg** operand is a longword that contains the processor register number. Execution may have register-specific side effects.

Notes

1 If the processor internal register does not exist, a reserved operand fault occurs.

2 A reserved instruction fault occurs if instruction execution is attempted in other than kernel mode.

3 A reserved operand fault occurs on a move from a write-only register.

BUG

Bugcheck

FORMAT

opcode *message.bx*

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

reserved instruction

opcodes

FEFF	BUGW	Bugcheck with word message identifier
FDFF	BUGL	Bugcheck with longword message identifier

DESCRIPTION

The hardware treats these opcodes as reserved to Digital and as faults. The VMS operating system treats them as requests to report software detected errors. The inline message identifier is zero extended to a longword (BUGW) and interpreted as a condition value (see the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*). If the process is privileged to report bugs, a log entry is made. If the process is not privileged, a reserved instruction is signaled.

EXAMPLES

```
BUGW                    ; Bugcheck with word message
.WORD    4              ;   identifier 4

BUGL                    ; Bugcheck with longword
.LONG    5              ;   message identifier 5
```