EK-VAXAR-RM-001

VAX-11 Architecture Reference Manual

20 May 1982

Revision 6.1

VAX-11 Architecture Reference Manual

digital

Revision 1, Sept, 1975
Revision 3, June, 1976
Revision 4, May, 1977
Revision 5, Feb, 1979
Revision 6, May, 1980
Revision 6.1, June,1981

ASSIST-11	DIBOL	KI10	RSTS
COMPUTER LABS	DIGITAL	KL10	RSX
COMSYST	DNC	LAB-8	RT-11
COMTEX	EDGRIN	LAB-K	RTS-8
DDT	EDUSYSTEM	MASSBUS	SABR
DEC	FLIP CHIP	OMNIBUS	SBI
DECnet	FOCAL	OS/8	TRAX
DECCOMM	GLC-8	PDP	TYPESET-8
DECUS	IDAC	PHA	TYPESET-10
DECsystem-10	IDACS	PS/8	TYPESET-11
DECsystem-20	INDAC	QUICKPOINT	UNIBUS
DECtape	KA10	RAD-8	VAX

PREFACE

The VAX-11 is a family of upward-compatible computer systems. It is a natural outgrowth of and is strongly compatible with the PDP-11 family. We believe that these systems represent a significant departure from traditional methods of computer design. VAX-11 represents the culmination of years of analysis of the needs of software, and compilers in particular.

For readers interested in just a summary of the family, please refer to the VAX Technical Summary. This manual explains the machine language programming and operation of any member of the VAX-11 family, for both instructional and reference purposes. Basically the manual defines in detail how the central processor functions, exactly what its instructions do, how it handles data, what its control and status information means, and what programming techniques and procedures must be employed to utilize it effectively. The programming is given in machine language, in that it uses only the basic instruction mnemonics and symbolic addressing defined by the assembler. The treatment relies neither on any other Digital software nor on any of the more sophisticated features of the assembler. Moreover, the manual is completely self-contained -- no prior knowledge of the assembler is required.

The text of the manual is devoted almost entirely to functional description and programming. Chapter 1 discusses the goals of the system and the notational conventions used throughout the manual. Chapter 2 defines the formats of the various forms of data and instructions. Chapter 3 discusses the addressing modes used in instructions. Chapter 4 gives the definition and detailed description of all instructions generally available to users of the system. Chapter 5 defines the memory management aspects of the system. Chapter 6 discusses the interrupt and exception handling in the system. Chapter 7 covers process structure and context switching. Chapter 8 defines those interactions between processor, memory, and I/O devices which are true of any member of the family. Chapter 9 defines the specifics of interacting with processor registers. Chapter 10 documents the PDP-11 Compatibility Mode of operation. Appendix A is a summary of the instructions, their operands, and the encoding. It is suitable to be used to construct an "instruction card".

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1-Feb-8Ø -- Rev 6

1.1 INTRODUCTION

VAX-11 represents a significant extension of the PDP-11 family architecture. It shares with the PDP-11 byte addressing, similar I/O and interrupt structures, and identical data formats. Although the instruction set is not strictly compatible with the PDP-11, it is related, and can be mastered easily by a PDP-11 programmer. Likewise the similarity enables straightforward manual conversion of existing PDP-11 programs to VAX-11. Existing user mode PDP-11 programs which do not need the extended features of VAX-11 can run unchanged in the PDP-11 compatibility mode provided in VAX-11.

As compared to the PDP-11, VAX-11 offers a greatly extended virtual address space, additional instructions and data types, and new addressing modes. Also provided is a sophisticated memory management and protection mechanism, and hardware assisted process scheduling and synchronization.

A number of specific goals guided the VAX-11 design:

1. Maximal compatibility with the PDP-11 consistent with a significant extension of the virtual address space, and a significant functional enhancement.

2. High bit efficiency. This is achieved by a wide range of data types and new addressing modes. PDP-11 programs naively translated to VAX-11 should not grow significantly in size; while programs redesigned to exploit VAX-11 should get smaller despite the extended virtual address space.

3. A systematic, elegant instruction set with orthogonality of operators, data types, and addressing modes. This enables the instruction set to be exploited easily, particularly by high level language processors.

4. Extensibility. The instruction set is designed so that new data types and operators can be included efficiently in a manner consistent with the currently defined operators and data types.

5. Range. The architecture should be suitable over the entire range of PDP-11 computer system implementations currently sold by Digital Equipment Corporation.

The VAX-11 Architecture Reference Manual describes the architecture of VAX-11 and applies to all implementations of VAX-11 systems.

1.2 TERMINOLOGY AND CONVENTIONS

1.2.1 Numbering

All numbers unless otherwise indicated are decimal. Where there is ambiguity, numbers other than decimal are indicated with the base in English following the number in parentheses (e.g., FF (hex)).

1.2.2 UNPREDICTABLE And UNDEFINED

Results specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE. Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing to stopping system operation. UNDEFINED operations must not cause the processor to hang i.e. reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions. Note the distinction between result and operation. Non-privileged software can not invoke UNDEFINED operations.

1.2.3 Ranges And Extents

Ranges are specified in English and are inclusive (e.g., a range of integers 0 through 4 includes the integers 0, 1, 2, 3, and 4.) Extents are specified by a pair of numbers separated by a colon and are inclusive (i.e. bits 7:3 specifies an extent of bits including bits 7, 6, 5, 4, and 3).

1.2.4 MBZ

Fields specified as MBZ (Must Be Zero) should never be filled by software with a non-zero value. If the processor encounters a non-zero value in a field specified as MBZ, a reserved operand fault or abort occurs (see Chapter 6, Exceptions and Interrupts) if that field is accessible to non-privileged software. MBZ fields that are accessible only to privileged software (kernel mode) may not be checked for non-zero value by some or all VAX-11 implementations. Non-zero values in MBZ fields accessible only to privileged software may produce UNDEFINED operation.

1.2.5 Reserved

Unassigned values of fields are reserved for future use. In many cases, some values are indicated as reserved to CSS/customers. Only these values should be used for non-standard applications. The values indicated as reserved to DEC and all MBZ fields are to be used only to extend the standard architecture in the future.

1.2.6 Figure Drawing Conventions

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

CHAPTER 2
BASIC ARCHITECTURE

29-Feb-8Ø -- Rev 6

2.1 ADDRESSING

The basic addressable unit in VAX-11 is the 8-bit byte. Virtual addresses are 32 bits long: hence the virtual address space is 2**32 (approximately 4.3 billion) bytes. Virtual addresses as seen by the program are translated into physical memory addresses by the memory management mechanism described in Chapter 5.

2.2 DATA TYPES

2.2.1 Byte

A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from the right Ø through 7:

```
 7              Ø
+---------------+
|               | :A
+---------------+
```

A byte is specified by its address A. When interpreted arithmetically, a byte is a twos complement integer with bits of increasing significance going Ø through 6 and bit 7 the sign bit. The value of the integer is in the range -128 through 127. For the purposes of addition, subtraction, and comparison, VAX-11 instructions also provide direct support for the interpretation of a byte as an unsigned integer with bits of increasing significance going Ø through 7. The value of the unsigned integer is in the range Ø through 255.

2.2.2 Word

A word is 2 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right Ø through 15:

```
         1
         5                              Ø
        +--------------------------------+
        |                                | :A
        +--------------------------------+
```

A word is specified by its address A, the address of the byte containing bit Ø. When interpreted arithmetically, a word is a twos complement integer with bits of increasing significance going Ø through 14 and bit 15 the sign bit. The value of the integer is in the range -32,768 through 32,767. For the purposes of addition, subtraction and comparison, VAX-11 instructions also provide direct support for the interpretation of a word as an unsigned integer with bits of increasing significance going Ø through 15. The value of the unsigned integer is in the range Ø through 65,535.

2.2.3 Longword

A longword is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right Ø through 31:

```
 3                                                                 Ø
 1
+-----------------------------------------------------------------+
|                                                                 | :A
+-----------------------------------------------------------------+
```

A longword is specified by its address A, the address of the byte containing bit Ø. When interpreted arithmetically, a longword is a twos complement integer with bits of increasing significance going Ø through 3Ø and bit 31 the sign bit. The value of the integer is in the range -2,147,483,648 through 2,147,483,647. For the purposes of addition, subtraction, and comparison, VAX-11 instructions also provide direct support for the interpretation of a longword as an unsigned integer with bits of increasing significance going Ø through 31. The value of the unsigned integer is in the range Ø through 4,294,967,295.

Note that the longword format is different from the longword format defined by the PDP-11 FP-11. In that format, bits of increasing significance go from 16 through 31 and Ø through 14. Bit 15 is the sign bit. Most DEC software and in particular PDP-11 FORTRAN and COBOL use the VAX-11 longword format.

2.2.4 Quadword

A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right Ø through 63:

```
 3
 1                                                        Ø
+----------------------------------------------------------+
|                                                          | :A
+----------------------------------------------------------+
|                                                          | :A+4
+----------------------------------------------------------+
 6                                                        3
 3                                                        2
```

A quadword is specified by its address A, the address of the byte containing bit Ø. When interpreted arithmetically, a quadword is a twos complement integer with bits of increasing significance going Ø through 62 and bit 63 the sign bit. The value of the integer is in the range -2**63 to 2**63-1. The quadword data type is not fully supported by VAX-11 instructions.

2.2.5 Octaword

A octaword is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right Ø through 127:

```
 3
 1                                                        Ø
+----------------------------------------------------------+
|                                                          | :A
+----------------------------------------------------------+
|                                                          | :A+4
+----------------------------------------------------------+
|                                                          | :A+8
+----------------------------------------------------------+
|                                                          | :A+12
+----------------------------------------------------------+
 1                                                        9
 2                                                        6
 7
```

A octaword is specified by its address A, the address of the byte containing bit Ø. When interpreted arithmetically, a octaword is a twos complement integer with bits of increasing significance going Ø through 126 and bit 127 the sign bit. The value of the integer is in the range -2**127 to 2**127-1. The octaword data type is not fully supported by VAX-11 instructions.

2.2.6 F_floating

A F_floating datum is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right Ø through 31.

```
 1 1
 5 4            7 6           Ø
+-+---------------+-------------+
|S|      exp      |  fraction   | :A
+-+-----------------------------+
|            fraction           | :A+2
+-------------------------------+
```

A F_floating datum is specified by its address A, the address of the byte containing bit Ø. The form of a F_floating datum is sign magnitude with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:Ø and 31:16 a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 16 through 31 and Ø through 6. The 8-bit exponent field encodes the values Ø through 255. An exponent value of Ø together with a sign bit of Ø, is taken to indicate that the F_floating datum has a value of Ø. Exponent values of 1 through 255 indicate true binary exponents of -127 through +127. An exponent value of Ø, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault (See Chapter 4 and 6). The value of a F_floating datum is in the approximate range .29*1Ø**-38 through 1.7*1Ø**38. The precision of a F_floating datum is approximately one part in 2**23, i.e., typically 7 decimal digits.

2.2.7 D_floating

A D_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right Ø through 63:

```
 1 1
 5 4            7 6           Ø
+-+---------------+-------------+
|S|      exp      |  fraction   | :A
+-+---------------+-------------+
|            fraction           | :A+2
+-------------------------------+
|            fraction           | :A+4
+-------------------------------+
|            fraction           | :A+6
+-------------------------------+
```

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

2.2.8 G_floating

A G_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 63:

```
 1 1
 5 4                  4 3     0
+-+-------------------+-------+
|S|   exp             | fract |  :A
+-+-------------------+-------+
|          fraction           |  :A+2
+-----------------------------+
|          fraction           |  :A+4
+-----------------------------+
|          fraction           |  :A+6
+-----------------------------+
```

A G_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a G_floating datum is sign magnitude with bit 15 the sign bit, bits 14:4 an excess 1024 binary exponent, and bits 3:0 and 63:16 a normalized 53-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11-bit exponent field encodes the values 0 through 2047. An exponent value of 0 together with a sign bit of 0, is taken to indicate that the G_floating datum has a value of 0. Exponent values of 1 through 2047 indicate true binary exponents of -1023 through +1023. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault (See Chapter 4 and 6). The value of a G_floating datum is in the approximate range .56*10**-308 through .9*10**308. The precision of a G_floating datum is approximately one part in 2**52, i.e., typically 15 decimal digits.

2.2.9 H_floating

A H_floating datum is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 127:

```
 1 1
 5 4                          Ø
+-+---------------------------+
|S|        exponent           | :A
+-+---------------------------+
|           fraction          | :A+2
+-----------------------------+
|           fraction          | :A+4
+-----------------------------+
|           fraction          | :A+6
+-----------------------------+
|           fraction          | :A+8
+-----------------------------+
|           fraction          | :A+1Ø
+-----------------------------+
|           fraction          | :A+12
+-----------------------------+
|           fraction          | :A+14
+-----------------------------+
```

A H_floating datum is specified by its address A, the address of the byte containing bit Ø. The form of a H_floating datum is sign magnitude with bit 15 the sign bit, bits 14:Ø an excess 16384 binary exponent, and bits 127:16 a normalized 113-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go 112 through 127, 96 through 111, 8Ø through 95, 64 through 79,48 through 63, 32 through 47, and 16 through 31. The 15-bit exponent field encodes the values Ø through 32767. An exponent value of Ø together with a sign bit of Ø, is taken to indicate that the H_floating datum has a value of Ø. Exponent values of 1 through 32767 indicate true binary exponents of -16383 through +16383. An exponent value of Ø, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault (See Chapter 4 and 6). The value of a H_floating datum is in the approximate range .84*1Ø**-4932 through .59*1Ø**4932. The precision of a H_floating datum is approximately one part in 2**112, i.e., typically 33 decimal digits.

2.2.1Ø Variable Length Bit Field

A variable bit field is Ø to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by 3 attributes: the address A of a byte, a bit position P which is the starting location of the field with respect to bit Ø of the byte at A, and a size S of the field. The specification of a bit field is indicated by the following where the field is the shaded area.

```
            P+S P+S-1                P P-1                      Ø
+--------------+-----------------------+-------------------------+
|              |///////////////////////|                         | :A
+--------------+-----------------------+-------------------------+
                S-1                    Ø
```

For bit strings in memory, the position is in the range -2**31 through 2**31-1 and is conveniently viewed as a signed 29-bit byte offset and a 3-bit bit-within-byte field:

```
 3
 1                                                      3 2     Ø
+------------------------------------------------------+-------+
|                      byte offset                      | bwb   |
+------------------------------------------------------+-------+
```

The sign extended 29-bit byte offset is added to the address A and the resulting address specifies the byte in which the field begins. The 3-bit bit-within-byte field encodes the starting position (Ø through 7) of the field within that byte. The VAX-11 field instructions provide direct support for the interpretation of a field as a signed or unsigned integer. When interpreted as a signed integer, it is twos complement with bits of increasing significance going Ø through S-2; bit S-1 is the sign bit. When interpreted as an unsigned integer, bits of increasing significance go from Ø to S-1. A field of size Ø has a value identically equal to Ø.

A variable bit field may be contained in 1 to 5 bytes. From a memory management point of view (Chapter 5) only the minimum number of bytes necessary to contain the field is actually referenced.

For bit fields in registers, the position is in the range Ø through 31. The position operand specifies the starting position (Ø through 31) of the field in the register. A variable bit field may be contained in 2 registers if the sum of position and size exceeds 32.

```
 3
 1      P P-1                                              Ø
+---------+-------------------------------------------------+
|/////////|                                                 | Rn
+-----------------------------------------------------------+
|                                              |////////////| R[n+1]
+----------------------------------------------+------------+
                                             P+S P+S-1
```

For further details on the specification of variable length bit fields see Chapter 4.

2.2.11 Character String

A character string is a contiguous sequence of bytes in memory. A character string is specified by 2 attributes: the address A of the first byte of the string, and the length L of the string in bytes. Thus the format of a character string is:

```
 7             Ø
+---------------+
|               | :A
+---------------+

       .
       .
       .

+---------------+
|               | :A+L-1
+---------------+
 7             Ø
```

The address of a string specifies the first character of a string. Thus "XYZ" is represented:

```
+---------------+
|   "X"         | :A
+---------------+
|   "Y"         | :A+1
+---------------+
|   "Z"         | :A+2
+---------------+
```

The length L of a string is in the range Ø through 65,535.

2.2.12 Trailing Numeric String

A trailing numeric string is a contiguous sequence of bytes in memory. The string is specified by 2 attributes : the address A of the first byte (most significant digit) of the string, and the length L of the string in bytes.

All bytes of a trailing numeric string, except the least significant digit byte, must contain an ASCII decimal digit character (Ø-9). The representation for the high order digits is:

digit	decimal	hex	ASCII character
Ø	48	3Ø	Ø
1	49	31	1
2	5Ø	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The highest addressed byte of a trailing numeric string represents an encoding of both the least significant digit and the sign of the numeric string. The VAX numeric string instructions support any encoding; however there are 3 preferred encodings used by DEC software. These are (1) unsigned numeric in which there is no sign and the least significant digit contains an ASCII decimal digit character, (2) zoned numeric, and (3) overpunched numeric. Because the overpunch format has been used by compilers of many manufacturers over many years, and because various card encodings are used, several variations in overpunch format have evolved. Typically, these alternate forms are accepted on input; the normal form is generated as the output for all operations. The valid representations of the digit and sign in each of the later two formats is:

Representation of Least Significant Digit and Sign

	Zoned Numeric Format			Overpunch Format			
digit	decimal	hex	ASCII char	decimal	hex	ASCII char norm	alt.
Ø	48	3Ø	Ø	123	7B	{	Ø [?
1	49	31	1	65	41	A	1
2	5Ø	32	2	66	42	B	2
3	51	33	3	67	43	C	3
4	52	34	4	68	44	D	4
5	53	35	5	69	45	E	5
6	54	36	6	7Ø	46	F	6
7	55	37	7	71	47	G	7
8	56	38	8	72	48	H	8
9	57	39	9	73	49	I	9
-Ø	112	7Ø	p	125	7D	}	] ! :
-1	113	71	q	74	4A	J	
-2	114	72	r	75	4B	K	
-3	115	73	s	76	4C	L	
-4	116	74	t	77	4D	M	
-5	117	75	u	78	4E	N	
-6	118	76	v	79	4F	O	
-7	119	77	w	8Ø	5Ø	P	
-8	12Ø	78	x	81	51	Q	
-9	121	79	y	82	52	R	

The length L of a trailing numeric string must be in the range Ø to 31 (Ø to 31 digits). The value of a Ø length string is identically Ø.

The address A of the string specifies the byte of the string containing the most significant digit. Digits of decreasing significance are assigned to increasing addresses. Thus "123" is represented:

```
      Zoned Format or Unsigned               Overpunch Format

        7     4 3     0                        7     4 3     0
       +-------+-------+                      +-------+-------+
       |   3   |   1   |  : A                 |   3   |   1   |  : A
       +-------+-------+                      +-------+-------+
       |   3   |   2   |  : A+1               |   3   |   2   |  : A+1
       +-------+-------+                      +-------+-------+
       |   3   |   3   |  : A+2               |   4   |   3   |  : A+2
       +-------+-------+                      +-------+-------+
```

and "-123" is represented :

```
          Zoned Format                         Overpunch Format

        7     4 3     0                        7     4 3     0
       +-------+-------+                      +-------+-------+
       |   3   |   1   |  : A                 |   3   |   1   |  : A
       +-------+-------+                      +-------+-------+
       |   3   |   2   |  : A+1               |   3   |   2   |  : A+1
       +-------+-------+                      +-------+-------+
       |   7   |   3   |  : A+2               |   4   |   C   |  : A+2
       +-------+-------+                      +-------+-------+
```

2.2.13 Leading Separate Numeric String

A leading separate numeric string is a contiguous sequence of bytes in memory. A leading separate numeric string is specified by 2 attributes: the address A of the first byte (containing the sign character), and a length L, which is the length of the string in digits and NOT the length of the string in bytes. The number of bytes in a leading separate numeric string is L+1.

The sign of a separate leading numeric string is stored in a separate byte. Valid sign bytes are:

Sign	decimal	hex	ASCII character
+	43	2B	+
+	32	20	<blank>
-	45	2D	-

The preferred representation for "+" is ASCII "+". All subsequent bytes contain an ASCII digit character:

digit	decimal	hex	ASCII character
Ø	48	3Ø	Ø
1	49	31	1
2	5Ø	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The length L of a leading separate numeric string must be in the range Ø to 31 (Ø to 31 digits). The value of a Ø length string is identically Ø.

The address A of the string specifies the byte of the string containing the sign. Digits of decreasing significance are assigned to bytes of increasing addresses. Thus "+123" is:

```
 7      4 3      Ø
+--------+--------+
|   2    |   B    |  : A
|--------+--------|
|   3    |   1    |  : A+1
|--------+--------|
|   3    |   2    |  : A+2
|--------+--------|
|   3    |   3    |  : A+3
+--------+--------+
```

and "-123" is:

```
 7      4 3      Ø
+--------+--------+
|   2    |   D    |  : A
|--------+--------|
|   3    |   1    |  : A+1
|--------+--------|
|   3    |   2    |  : A+2
|--------+--------|
|   3    |   3    |  : A+3
+--------+--------+
```

2.2.14 Packed Decimal String

A packed decimal string is a contiguous sequence of bytes in memory. A packed decimal string is specified by 2 attributes: the address A of the first byte of the string and a length L which is the number of digits in the string and NOT the length of the string in bytes. The bytes of a packed decimal string are divided into 2 4-bit fields (nibbles) which must contain decimal digits except the low nibble (bits 3:Ø) of the last (highest addressed) byte which must contain a sign. The representation for the digits and sign is:

digit or sign	decimal	hex
Ø	Ø	Ø
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
+	1Ø,12,14 or 15	A,C,E, or F
-	11 or 13	B, or D

The preferred sign representation is 12 for "+" and 13 for "-". The length L is the number of digits in the packed decimal string (not counting the sign) and must be in the range Ø through 31. When the number of digits is odd, the digits and the sign fit in L/2 (integer part only) + 1 bytes. When the number of digits is even, it is required that an extra "Ø" digit appear in the high nibble (bits 7:4) of the first byte of the string. Again the length in bytes of the string is L/2 + 1.

The address A of the string specifies the byte of the string containing the most significant digit in its high nibble. Digits of decreasing significance are assigned to increasing byte addresses and from high nibble to low nibble within a byte. Thus "+123" has length 3 and is represented:

```
 7      4 3      Ø
+-------+-------+
|   1   |   2   |  : A
+-------+-------+
|   3   |  12   |  : A + 1
+-------+-------+
```

and "-12" has length 2 and is represented:

```
 7      4 3      0
+--------+--------+
|   0    |   1    |  : A
+--------+--------+
|   2    |   13   |  : A + 1
+--------+--------+
```

2.3 PROCESSOR STATE

The processor state consists of that portion of a process's state which, while the process is executing, is stored in processor registers rather than memory. The processor state described here consists of that accessible to non-privileged software. Certain additional processor state is described in Chapters 5, 6, and 7.

The non-privileged processor state includes 16 32-bit general purpose registers denoted Rn where n is in the range 0 through 15 and a 16-bit processor status word (PSW). Where there is ambiguity (e.g., n is an arithmetic expression) the notation R[n] is also used to denote the register. The general purpose registers are used for temporary storage, accumulators, index registers, and base registers. A register containing an address is termed a base register. A register containing an address offset (in multiples of operand size, see Chapter 3) is termed an index register.

The bits of a register are numbered from the right 0 through 31:

```
 3
 1                                                          0
+------------------------------------------------------------+
|                                                            | :Rn
+------------------------------------------------------------+
```

Certain of the registers are assigned special meaning by the VAX-11 architecture:

1. R15 is the program counter (PC). PC contains the address of the next instruction byte of the program.

2. R14 is the stack pointer (SP). SP contains the address of the top of the processor defined stack.

3. R13 is the current frame pointer (FP). The VAX-11 procedure call convention (see VAX/VMS Run Time Library Reference Manual) builds a data structure on the stack called a stack frame. FP contains the address of the base of this data structure.

4. R12 is the argument pointer (AP). The VAX-11 procedure call convention uses a data structure termed an argument list. AP contains the address of the base of this data structure.

Note that these registers are all used as base registers. The assignment of special meaning to these registers does not generally preclude their use for other purposes. However, as will be seen in Chapter 3, PC cannot be used as an accumulator, temporary, or index register.

When a datum of type byte, word, longword, or F_floating is stored in a register, the bit numbering in the register corresponds to the numbering in memory. Hence a byte is stored in register bits 7:0, a word in register bits 15:0, and longword or F_floating, in register bits 31:0. A byte or word written to a register writes only bits 7:0 and 15:0 respectively; the other bits are unaffected. A byte or word read from a register reads only bits 7:0 and 15:0 respectively; the other bits are ignored.

When a quadword, D_floating or G_floating datum is stored in a register R[n], it is actually stored in 2 adjacent registers R[n] and R[n+1]. Because of restrictions on the specification of PC (see Chapter 3) wraparound from PC to R0 is UNPREDICTABLE. Bits 31:0 of the datum are stored in bits 31:0 of register R[n] and bits 63:32 of the datum are stored in bits 31:0 of register R[n+1].

When an octaword or a H_floating datum is stored in register R[n], it is actually stored in adjacent registers R[n], R[n+1], R[n+2], and R[n+3]. Because of restrictions on the specification of PC (see Chapter 3) wraparound from PC to R0 is UNPREDICTABLE. Bits 31:0 of the datum are stored in bits 31:0 of register R[n], bits 63:32 in bits 31:0 of register R[n+1], bits 95:64 in bits 31:0 of register R[n+2], and bits 127:96 in bits 31:0 of register R[n+3].

With one restriction, a variable length bit field may be specified in the registers: the starting bit position P must be in the range 0 through 31. As for quadword, D_floating, and G_floating, a pair of registers R[n] and R[n+1] is treated as a 64-bit register with bits 31:0 in register R[n] and bit 63:32 in register R[n+1].

None of the string data types stored in registers can be processed by the VAX-11 string instructions. Thus there is no architectural specification of the representation of strings in registers.

2.4 PROCESSOR STATUS WORD

The processor status word (PSW) contains the condition codes which give information on the results produced by previous instructions and the exception enables which control the processor action on certain exception conditions (see Chapter 6). The format of the PSW is:

```
 1
 5             8 7 6 5 4 3 2 1 0
+---------------+-+-+-+-+-+-+-+-+
|               |D|F|I| | | | | |
|     MBZ       |V|U|V|T|N|Z|V|C|
+---------------+-+-+-+-+-+-+-+-+
```

The condition codes are UNPREDICTABLE when they are affected by UNPREDICTABLE results. The VAX-11 procedure call instructions (See Chapter 4) conditionally set the IV and DV enables, clear the FU enable, and leave the T enable unchanged at procedure entry.

2.4.1 C Bit

When set, the C (carry) condition code bit indicates the last instruction which affected C had a carry out of the most significant bit of the result or a borrow into the most significant bit. When C is clear, there was no carry or borrow.

2.4.2 V Bit

When set, the V (overflow) condition code bit indicates that the last instruction which affected V produced a result whose magnitude was too large to be properly represented in the operand which received the result or there was a conversion error. When V is clear, there was no overflow or conversion error.

2.4.3 Z Bit

When set, the Z (zero) condition code indicates that the last instruction which affected Z produced a result which was 0. When Z is clear, the result was non-zero.

2.4.4 N Bit

When set, the N (negative) condition code bit indicates that the last instruction which affected N produced a result which was negative. When N is clear, the result was positive (or zero).

2.4.5 T Bit

When set at the beginning of an instruction, the T (trace) bit causes the TP bit in the Processor Status Longword to be set (see Chapter 6). When TP is set at the end of an instruction, a trace fault is taken before the execution of the next instruction. See Chapter 6 for additional information on the trace fault.

2.4.6 IV Bit

When set, the IV (integer overflow) bit forces an integer overflow trap after execution of an instruction which produced an integer result that overflowed or had a conversion error. When IV is clear, no integer overflow trap occurs. (However, the condition code V bit is still set.)

2.4.7 FU Bit

When set, the FU (floating underflow) bit forces a floating underflow fault if the result of a floating point instruction is too small in magnitude to be represented in the result operand. When FU is clear, no underflow fault occurs.

2.4.8 DV Bit

When set, the DV (decimal overflow) bit forces a decimal overflow trap after execution of an instruction which produced an overflowed decimal (numeric string, or packed decimal) result or had a conversion error. When DV is clear, no trap occurs. (However, the condition code V bit is still set.)

2.5 PERMANENT EXCEPTION ENABLES

The processor action on certain exception conditions is not controlled by bits in the PSW. Traps or faults always result from these exception conditions.

2.5.1 Divide By Zero

A divide by zero trap is forced after the execution of integer, or decimal division instruction which has a zero divisor. A fault occurs on a floating division instruction which has a zero divisor.

2.5.2 Floating Overflow

A floating overflow fault is forced after the execution of a floating point instruction which produced a result too large to be represented in the result operand.

2.6 INSTRUCTION FORMAT

VAX-11 has a variable length instruction format. An instruction specifies an operation and Ø to 6 operands. An operation specifier is termed an opcode. Depending on the instruction the opcode is 1 or 2 bytes long. An operand specifier indicates the addressing mode used to access the operand and may be 1 or 2 bytes. An operand specifier may be followed by a specifier extension, an address, or immediate data. The format of an n operand instruction is:

```
opcode
operand specifier 1
specifier extension, address, or immediate data 1 (if needed)
operand specifier 2
        .
        .
        .
operand specifier n
specifier extension, address, or immediate data n (if needed)
```

See Chapter 3 for a full description of addressing modes. See Chapter 4 for a definition of the instructions. See Appendix A for a summary of all operands, instructions, and their binary assignments.

2.7 SEPARATION OF PROCEDURE AND DATA

The VAX-11 architecture encourages (and provides the mechanisms to facilitate) separation of procedure (instructions) and writable data. Procedures may not write data which is to be subsequently executed as an instruction without an intervening REI instruction being executed (See Chapter 6) or an intervening context switch occurring (See Chapter 7). If no REI or context switch occurs between a procedure writing data as instructions to be executed, and those instructions being executed, the instructions executed are UNPREDICTABLE.

2.8 I/O STRUCTURE

Generally, the VAX-11 I/O structure closely follows that of the PDP-11. An I/O device controller is defined by a set of registers. The registers are assigned addresses in the physical address space. Commands are issued to I/O controllers by the processor writing these registers; controllers return status by writing these registers and the processor subsequently reading them. Since the registers have memory addresses, ordinary instructions can read or write them; no special I/O instructions are needed. The normal memory management mechanism controls access to device controller registers.

2.9 INTERRUPT STRUCTURE

A VAX-11 processor provides a 32 level vectored priority interrupt system. This is described in detail in Chapter 6.

CHAPTER 3
INSTRUCTION FORMATS AND ADDRESSING MODES

5-May-8Ø -- Rev 7

3.1 OPCODE FORMATS

An instruction is specified by the byte address A of its opcode:

```
 7              Ø
+---------------+
|    opcode     | :A
+---------------+
```

The opcode may extend over 2 bytes; the length depends on the contents of the byte at address A. If, and only if, the value of the byte is FC (hex) through FF (hex) is the opcode 2 bytes long:

```
 1
 5             8 7              Ø
+---------------+---------------+
|    opcode     |   FC - FF     | :A
+---------------+---------------+
```

3.2 OPERAND SPECIFIERS

Each instruction takes a specific sequence of operand specifier types. An operand specifier type conceptually has two components: the access type and the data type.

The access types include:

1. Read - the specified operand is read only.
2. Write - the specified operand is written only.
3. Modify - the specified operand is read, potentially modified, and written. This is not done under a memory interlock.
4. Address - the address of the specified operand in the form of a longword is the actual instruction operand. The specified operand is not accessed directly although the instruction may subsequently use the address to access that operand.
5. Variable bit field base address - same as address access type except for register mode. In register mode, the field is contained in register n designated by the operand specifier (or register n+1 concatenated with register n). This access type is a special variant of the address access type.
6. Branch - no operand is accessed. The operand specifier itself is a branch displacement.

Types 1 - 5 are termed general mode addressing and are discussed in Section 3.4. Type 6 is termed branch mode addressing and is discussed in Section 3.6.

The data types include:

1. Byte
2. Word
3. Longword and F_floating which are equivalent for addressing mode considerations.
4. Quadword, and D_floating and G_floating which are similarly equivalent.
5. Octaword and H_floating which are also similarly equivalent.

For the address and branch access types which do not directly reference operands, the data type indicates:

1. Address - the operand size to be used in the address calculation in autoincrement, autodecrement, and index modes.

2. Branch - the size of the branch displacement.

3.3 NOTATION

To describe the addressing modes the following is used:

+	- addition
-	- subtraction
*	- multiplication
<-	- is replaced by
=	- is defined as
'	- concatenation
Rn or R[n]	- the contents of register n
PC or SP	- the contents of register 15 or 14 respectively

NOTE

In the formal descriptions of the addressing modes Rn or PC, for example, always means the contents of register n or register 15. When there is no ambiguity, Rn or PC, for example, is often used in text as the name of register n or register 15.

(x)	- the contents of a location in memory whose address is x.
{ }	- arithmetic parentheses used to indicate precedence
SEXT(x)	- x is sign extended to size of operand needed
ZEXT(x)	- x is zero extended to size of operand needed
OA	- operand address
!	- comment delimiter

Each general mode addressing description includes the definition of the operand address, and the specified operand. For operand specifiers of address access type, the operand address is the actual instruction operand; for other access types the specified operand is the instruction operand. The branch mode addressing description includes the definition of the branch address.

3.4 GENERAL MODE ADDRESSING FORMATS

3.4.1 Register Mode

The operand specifier format is:

```
 7     4 3     Ø
+-------+-------+
|   5   |  Rn   |
+-------+-------+
```

No specifier extension follows.

In register mode addressing the operand is the contents of register n (or register n+1 concatenated with register n for quadword, D_floating, and certain field operands):

```
operand = Rn                          !if one register
          or
          R[n+1]'Rn                   !if two registers
          or
          R[n+3]'R[n+2]'R[n+1]'Rn     !if four registers
```

Because registers do not have memory addresses, the operand address is not defined, and register mode may not be used for operand specifiers of address access type (except in the case of the base address for variable bit field instructions, see Chapter 4). If it is, an illegal addressing mode fault results (See Chapter 6). PC may not be used in register mode addressing. If PC is read, the value read is UNPREDICTABLE. If PC is written, the next instruction executed or the next operand specified is UNPREDICTABLE. Likewise, SP may not be used in register mode addressing for an operand which takes two adjacent registers. Again, if it is, the results are UNPREDICTABLE in the same fashion. If PC is used in register mode for a write access type operand which takes 2 adjacent registers, the contents of RØ are UNPREDICTABLE. If R12, R13, SP, or PC are used in register mode addressing for an operand which takes four adjacent registers, the results are UNPREDICTABLE. If PC is used in register mode for a write access type operand which requires 4 adjacent registers, the contents of RØ, R1, and R2 are UNPREDICTABLE. Likewise, if R13 is used in register mode for a write access type operand which takes 4 adjacent registers, the contents of RØ are UNPREDICTABLE; and, if SP is used in register mode for a write access type operand which takes 4 adjacent registers, the contents of RØ and R1 are UNPREDICTABLE.

The assembler notation for register mode is Rn.

3.4.2 Register Deferred Mode

The operand specifier format is:

```
 7     4 3     0
+-------+-------+
|   6   |  Rn   |
+-------+-------+
```

No specifier extension follows.

In register deferred mode addressing, the address of the operand is the contents of register n:

```
OA = Rn

operand = (OA)
```

PC may not be used in register deferred mode addressing. If it is, the address of the operand (and whether the operand is written if it is of modify or write access type) is UNPREDICTABLE.

The assembler notation for register deferred mode is (Rn).

3.4.3 Autoincrement Mode

The operand specifier format is:

```
 7     4 3     0
+-------+-------+
|   8   |  Rn   |
+-------+-------+
```

No specifier extension follows. If Rn denotes PC, immediate data follows, and the mode is termed immediate mode.

In autoincrement mode addressing, the address of the operand is the contents of register n. After the operand address is determined, the size of the operand in bytes (1 for byte; 2 for word; 4 for longword and F_floating; 8 for quadword, G_floating and D_floating; and 16 for octaword, and H_floating)is added to the contents of register n and the contents of register n is replaced by the result:

```
OA = Rn

Rn <- Rn + size

operand = (OA)
```

Immediate mode may not be used for operands of modify or write access type. If immediate mode is used for an operand of modify access type, the value of the data read is UNPREDICTABLE. If immediate mode is used for an operand of modify or write access type, the address at which the operand is written (and whether it is written) is UNPREDICTABLE.

The assembler notation for autoincrement mode is (Rn)+. For immediate mode the notation is I^#constant where constant is the immediate data which follows.

3.4.4 Autoincrement Deferred Mode

The operand specifier format is:

```
 7      4 3      0
+-------+-------+
|   9   |  Rn   |
+-------+-------+
```

No specifier extension follows. If Rn denotes PC, a longword address follows, and the mode is termed absolute mode.

In autoincrement deferred mode addressing, the address of the operand is the contents of a longword whose address is the contents of register n. After the operand address is determined, 4 (the size in bytes of a longword address) is added to the contents of register n and the contents of register n is replaced by the result:

```
OA = (Rn)

Rn <- Rn + 4

operand = (OA)
```

The assembler notation for autoincrement deferred mode is @(Rn)+. For absolute mode the notation is @#address where address is the longword which follows.

3.4.5 Autodecrement Mode

The operand specifier format is:

```
 7      4 3      0
+--------+--------+
|   7    |   Rn   |
+--------+--------+
```

No specifier extension follows.

In autodecrement mode addressing, the size of the operand in bytes (1 for byte; 2 for word; 4 for longword and F_floating; 8 for quadword, G_floating and D_floating; and 16 for octaword, and H_floating)is subtracted from the contents of register n and the contents of register n are replaced by the result. The updated contents of register n is the address of the operand:

```
Rn  <- Rn  - size

OA = Rn

operand = (OA)
```

PC may not be used in autodecrement mode. If it is, the address of the operand (and whether the operand is written if it is of modify or write access type) is UNPREDICTABLE and the next instruction executed or the next operand specified is UNPREDICTABLE.

The assembler notation for autodecrement mode is -(Rn).

3.4.6 Displacement Mode

There are 3 operand specifier formats:

```
          7     4 3     0
         +-------+-------+
1.       |  10   |  Rn   |
         +-------+-------+
```

The specifier extension is a signed byte displacement, which follows the operand specifier. This is termed byte displacement mode.

```
          7     4 3     0
         +-------+-------+
2.       |  12   |  Rn   |
         +-------+-------+
```

The specifier extension is a signed word displacement, which follows the operand specifier. This is termed word displacement mode.

```
          7     4 3     0
         +-------+-------+
3.       |  14   |  Rn   |
         +-------+-------+
```

The specifier extension is a longword displacement, which follows the operand specifier. This is termed longword displacement mode.

In displacement mode addressing, the displacement (after being sign extended to 32 bits if it is byte or word) is added to the contents of register n and the result is the operand address:

```
        OA = Rn + SEXT(displ)           !if byte or word displacement
             or
             Rn + displ                 !if longword displacement

        operand = (OA)
```

If Rn denotes PC, the updated contents of PC is used. The updated contents of PC is the address of the first byte beyond the specifier extension.

The assembler notation for byte, word, and long displacement mode is B^D(Rn), W^D(Rn), and L^D(Rn) respectively where D = displ.

3.4.7 Displacement Deferred Mode

There are 3 operand specifier formats:

```
           7      4 3      0
          +-------+-------+
1.        |  11   |  Rn   |
          +-------+-------+
```

The specifier extension is a signed byte displacement, which follows the operand specifier. This is termed byte displacement deferred mode.

```
           7      4 3      0
          +-------+-------+
2.        |  13   |  Rn   |
          +-------+-------+
```

The specifier extension is a signed word displacement, which follows the operand specifier. This is termed word displacement deferred mode.

```
           7      4 3      0
          +-------+-------+
3.        |  15   |  Rn   |
          +-------+-------+
```

The specifier extension is a longword displacement, which follows the operand specifier. This is termed longword displacement deferred mode.

In displacement deferred mode addressing, the displacement (after being sign extended to 32 bits if it is byte or word) is added to the contents of register n and the result is the address of a longword whose contents is the operand address:

```
        OA = (Rn + SEXT(displ))          !if byte or word displacement
             or
             (Rn + displ)                !if longword  displacement

        operand = (OA)
```

If Rn denotes PC, the updated contents of the PC is used. The updated contents of PC is the address of the first byte beyond the specifier extension.

The assembler notation for byte, word, and longword displacement deferred mode is @B^D(Rn), @W^D(Rn), and @L^D(Rn) respectively where D = displ.

3.4.8 Literal Mode

The operand specifier format is:

```
  7 6 5         0
+---+-----------+
| 0 |   literal |
+---+-----------+
```

No specifier extension follows.

For operands of data type byte, word, longword, quadword, octaword the operand is the zero extension of the 6-bit literal field:

```
operand = ZEXT(literal)
```

Thus for these data types, literal mode may be used for values in the range 0 through 63.

For operands of data type F_floating, G_floating, D_floating, and H_floating, the 6-bit literal field is composed of 2 3-bit fields:

```
 5   3 2   0
+-----+-----+
| exp | fra |
+-----+-----+
```

where exp is exponent and fra is fraction. The exp and fra fields are used to form a F_floating or D_floating operand as follows:

```
 1 1
 5 4              7 6   4 3     0
+-+--------------+-----+-------+
|0|    128 + exp | fra |   0   |
+-+--------------+-------------+
|              0               | :A+2
+------------------------------+
|              0               | :A+4
+------------------------------+
|              0               | :A+6
+------------------------------+
```

where bits 63:32 are not present in a F_floating operand.

The exp and fra fields are used to form a G_floating operand as follows:

```
 1 1
 5 4                   4 3   1 0
+-+--------------------+-----+-+
|0|    1024 + exp      | fra |0|
+-+--------------------+-----+-+
|             0                 | :A+2
+-------------------------------+
|             0                 | :A+4
+-------------------------------+
|             0                 | :A+6
+-------------------------------+
```

The exp and fra fields are used to form a H_floating operand as follows:

```
 1 1
 5 4                           0
+-+-----------------------------+
|0|    16384 + exp              |
+-+---+-------------------------+
| fra |        0                | :A+2
+-----+-------------------------+
|              0                | :A+4
+-------------------------------+
|              0                | :A+6
+-------------------------------+
|              0                | :A+8
+-------------------------------+
|              0                | :A+10
+-------------------------------+
|              0                | :A+12
+-------------------------------+
|              0                | :A+14
+-------------------------------+
```

The range of values available is given in the following table:

E   F -->
|
v

	0	1	2	3	4	5	6	7
0	1/2	9/16	5/8	11/16	3/4	13/16	7/8	15/16
1	1	1 1/8	1 1/4	1 3/8	1 1/2	1 5/8	1 3/4	1 7/8
2	2	2 1/4	2 1/2	2 3/4	3	3 1/4	3 1/2	3 3/4
3	4	4 1/2	5	5 1/2	6	6 1/2	7	7 1/2
4	8	9	10	11	12	13	14	15
5	16	18	20	22	24	26	28	30
6	32	36	40	44	48	52	56	60
7	64	72	80	88	96	104	112	120

Table 1. Floating Literals

Because there is no operand address, literal mode addressing may not be used for operand specifiers of address access type. Literal mode addressing may also not be used for operand specifiers of write or modify access type. If literal mode is used for operand specifiers of either address, modify, or write access type, an illegal addressing mode fault results (see Chapter 6).

Literal mode addressing is a very efficient way of specifying integer constants in the range Ø to 63 and the floating point constants given in Table 1. Literal values outside the indicated range may be obtained by autoincrement mode using PC (immediate mode).

The assembler notation for literal mode is S^#literal.

3.4.9 Index Mode

The operand specifier format is:

```
 1
 5            8 7     4 3     Ø
+--------------+-------+-------+
|              |   4   |  Rx   |
+------------------------------+
```

Bits 15:8 contain a second operand specifier (termed the base operand specifier) for any of the addressing modes except register, literal or index. The specification of register, literal, or index addressing mode results in an illegal addressing mode fault (see Chapter 6). If the base operand specifier requires a specifier extension, it immediately follows. The base operand specifier is subject to the same restrictions as would apply if it were used alone. If the use of some particular specifier is illegal (i.e., causes a fault or UNPREDICTABLE behavior) under some circumstances, then that specifier is similarly illegal as a base operand specifier in index mode under the same circumstances.

The operand to be specified by index mode addressing is termed the primary operand. The base operand specifier is used normally to determine an operand address. This address is termed the base operand address (BOA). The address of the primary operand specified is determined by multiplying the contents of the index register x by the size of the primary operand in bytes (1 for byte; 2 for word; 4 for longword and F_floating; 8 for quadword, D_floating and G_floating; and 16 for octaword, and H_floating), adding BOA, and taking the result:

```
OA = BOA + {size * (Rx)}

operand = (OA)
```

If the base operand specifier is for autoincrement or autodecrement mode the increment or decrement size is the size in bytes of the primary operand.

Index mode addressing permits very general and efficient accessing of arrays. The base address of the array is determined by the operand address caculation of the base operand specifier. The contents of the index register is taken as a logical index into the array. The logical index is converted into a real (byte) offset by multiplying the contents of the index register by the size of the primary operand in bytes.

Certain restrictions are placed on the index register x. PC cannot be used as an index register. If it is, a reserved addressing mode fault occurs (see Chapter 6). If the base operand specifier is for an addressing mode which results in register modification (i.e. autoincrement mode, autodecrement mode, or autoincrement deferred mode), the same register cannot be the index register. If it is, the primary operand address is UNPREDICTABLE.

The names of the addressing modes resulting from index mode addressing are formed by adding the suffix "indexed" to the addressing mode of the base operand specifier. The following gives the names and assembler notation. The index register is designated Rx to distinguish it from the register Rn in the base operand specifier.

1. register deferred indexed - (Rn)[Rx]

2. autoincrement indexed - (Rn)+[Rx]

   or immediate indexed - I^#constant[Rx] which is recognized by the assembler but is not generally useful. Note that the operand address is independent of the value of constant.

3. autoincrement deferred indexed - @(Rn)+[Rx]

   or absolute indexed - @#address[Rx]

4. autodecrement indexed - -(Rn)[Rx]

5. byte, word, or longword displacement indexed - B^D(Rn)[Rx],W^D(Rn)[Rx], or L^D(Rn)[Rx]

6. byte, word, or longword displacement deferred indexed - @B^D(Rn)[Rx],@W^D(Rn)[Rx], or @L^D(Rn)[Rx]

3.5 SUMMARY OF GENERAL MODE ADDRESSING

3.5.1 General Register Addressing

```
 7     4 3     0
+-------+-------+
| mode  |  reg  |
+-------+-------+
```

Hex	Dec	Name	Assembler	r	m	w	a	v	PC	SP	AP& FP	Index- able
0-3	0-3	literal	S^#literal	y	f	f	f	f	-	-	-	f
4	4	indexed	i[Rx]	y	y	y	y	y	f	y	y	f
5	5	register	Rn	y	y	y	f	y	u	uq	uo	f
6	6	register deferred	(Rn)	y	y	y	y	y	u	y	y	y
7	7	autodecrement	-(Rn)	y	y	y	y	y	u	y	y	ux
8	8	autoincrement	(Rn)+	y	y	y	y	y	p	y	y	ux
9	9	autoincrement deferred	@(Rn)+	y	y	y	y	y	p	y	y	ux
A	10	byte displacement	B^D(Rn)	y	y	y	y	y	p	y	y	y
B	11	byte displacement deferred	@B^D(Rn)	y	y	y	y	y	p	y	y	y
C	12	word displacement	W^D(Rn)	y	y	y	y	y	p	y	y	y
D	13	word displacement deferred	@W^D(Rn)	y	y	y	y	y	p	y	y	y
E	14	longword displacement	L^D(Rn)	y	y	y	y	y	p	y	y	y
F	15	longword displacement deferred	@L^D(Rn)	y	y	y	y	y	p	y	y	y

3.5.2 Program Counter Addressing (reg=15)

```
 7     4 3 2 1 0
+-------+-+-+-+-+
| mode  |1 1 1 1|
+-------+-+-+-+-+
```

Hex	Dec	Name	Assembler	r	m	w	a	v	PC	SP	Indexable?
8	8	immmediate	I^#constant	y	u	u	y	y	-	-	y
9	9	absolute	@#address	y	y	y	y	y	-	-	y
A	10	byte relative	B^address	y	y	y	y	y	-	-	y
B	11	byte relative deferred	@B^address	y	y	y	y	y	-	-	y
C	12	word relative	W^address	y	y	y	y	y	-	-	y
D	13	word relative deferred	@W^address	y	y	y	y	y	-	-	y
E	14	long word relative	L^address	y	y	y	y	y	-	-	y
F	15	long word relative deferred	@L^address	y	y	y	y	y	-	-	y

Key to 3.5.1 and 3.5.2

D -	displacement
i -	any indexable addressing mode
- -	logically impossible
f -	reserved addressing mode fault
p -	Program Counter addressing
u -	UNPREDICTABLE
uq -	UNPREDICTABLE for quad, octa, D_floating, G_floating, and H_floating (and field if position + size greater than 32)
uo -	UNPREDICTABLE for octa, and H format
ux -	UNPREDICTABLE for index register same as base register
y -	yes, always valid addressing mode
r -	read access
m -	modify access
w -	write access
a -	address access
v -	field access

3.6 BRANCH MODE ADDRESSING FORMATS

There are 2 operand specifier formats:

```
          7             Ø
         +---------------+
1.       |     displ     |
         +---------------+
```

The operand specifier is a signed byte displacement.

```
         1
         5                             Ø
        +-------------------------------+
2.      |            displ              |
        +-------------------------------+
```

The operand specifier is a signed word displacement.

In branch displacement addressing, the byte or word displacement is sign extended to 32 bits and added to the updated contents of PC. The updated contents of PC is the address of the first byte beyond the operand specifier. The result is the branch address A:

```
        A = PC + SEXT(displ)
```

The assembler notation for byte and word branch displacement addressing is A where A is the branch address. Note that the branch address and not the displacement is used.

3.7 OPERAND SPECIFIER CONVENTIONS

The following 3 steps are performed by each instruction:

1. Each operand specifier in order of instruction stream occurrence is treated as follows:

    a. If read access type: evaluate the operand address, read the operand, and save it.

    b. If write access type: evaluate the operand address and save it.

    c. If modify access type: evaluate the operand address and save it; read the operand and save it.

    d. If address access type: evaluate the address and save it.

    e. If branch access type: save the operand specifier.

2. Perform the operation indicated by the instruction.

3. Store the result(s) using the saved addresses in the order indicated by the occurrence of operand specifiers in the instruction stream.

NOTE

The string (character, zoned decimal, and packed decimal) instructions are an exception to 2. and 3. in that partial results are stored before the instruction operation is completed. The variable bit field instructions treat the position, size, and base address operand specifiers as the specification of an implied field operand specifier (see Appendix A). If multiple exceptions occur during 1. and 2., the order in which they are taken is UNPREDICTABLE. This can occur, for example, in a floating point instruction whose destination operand specifier of write access type uses a reserved addressing mode and the operation results in an overflow fault.

The implications of these conventions are:

1. Autoincrement and autodecrement operations occur as the operand specifiers are processed, and subsequent operand specifiers use the updated contents of registers modified by those operations.

2. Other than as indicated by 1, all input operands are read, and all addresses of output operands computed before any results of the instruction are stored.

3. An operand of modify access type is not read, modified, and written as an indivisible operation; therefore, modify access type operands cannot be used for synchronization. (For synchronization instructions, See Chapter 8.)

4. If an instruction references two operands of write or modify access type at the same address, the first will be overwritten by the second.

CHAPTER 4
INSTRUCTIONS

12-Feb-82 -- Rev 7

4.1 INSTRUCTION SET

This chapter describes the instructions generally used by all software across all implementations of the VAX-11 architecture. Certain instructions which are specific to specialized portions of the VAX-11 architecture (e.g., memory management, interrupts and exceptions, process dispatching, and processor registers) and are generally used by privileged software are described in the chapters describing those portions of the architecture. A concise list of instructions and opcode assignments appears in Appendix A.

4.1.1 Instruction Descriptions

The instruction set is divided into 12 major sections:

1. Integer arithmetic and logical
2. Address
3. Variable length bit field
4. Control
5. Procedure call
6. Miscellaneous
7. Queue
8. Floating point
9. Character string

10. Cyclic Redundancy Check

11. Decimal string

12. Edit

Within each major section, instructions which are closely related are combined into groups and described together. The instruction group description is composed of the following:

1. The group name.

2. The format of each instruction in the group. This gives the name and type of each instruction operand specifier and the order in which it appears in memory. Operand specifiers from left to right appear in increasing memory addresses.

3. The operation of the instruction.

4. The effect on condition codes.

5. Exceptions specific to the instruction. Exceptions which are generally possible for all instructions (e.g., illegal or reserved addressing mode, T-bit, memory management violations, etc.) are not listed.

6. The opcodes, mnemonics, and names of each instruction in the group. The opcodes are given in hex.

7. A description in English of the instruction.

8. Optional notes on the instruction and programming examples.

4.1.2 Operand Specifier Notation

Operand specifiers are described in the following way:

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

where:

1. Name is a suggestive name for the operand in the context of the instruction. The name is often abbreviated.

2. Access type is a letter denoting the operand specifier access type:

   a - Calculate the effective address of the specified operand. Address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>; i.e. size to be used in autoincrement, autodecrement, and indexing.

   b - No operand reference. Operand specifier is a branch displacement. Size of branch displacement is given by <data type>.

   m - Operand is read, potentially modified and written. Note that this is NOT an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessability (See Chapter 5).

   r - Operand is read only.

   v - Calculate the effective address of the specified operand. If the effective address is in memory, the address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>. If the effective address is Rn, the operand is in Rn or R[n+1]'Rn.

   w - Operand is written only.

3. Data type is a letter denoting the data type of the operand:

   b - byte

   d - D_floating

f - F_floating

g - G_floating

h - H_floating

l - longword

o - octaword

q - quadword

w - word

x - first data type specified by instruction

y - second data type specified by instruction

4.1.3 Operation Description Notation

The operation of each instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax. No attempt is made to define the syntax formally, it is assumed to be familiar to the reader. The notation used is an extension of that introduced in Chapter 3.

+ - addition

- - subtraction, unary minus

* - multiplication

/ - division (quotient only)

** - exponentiation

' - concatenation

<- - is replaced by

= - is defined as

Rn or R[n] - contents of register Rn

PC, SP, FP, or AP - the contents of register R15, R14, R13, or R12 respectively

PSW - the contents of the processor status word

PSL - the contents of the processor status long word

(x) - contents of memory location whose address is x (x)+ - contents of memory location whose address is x; x incremented by the size of operand referenced at x

-(x) - x decremented by size of operand to be referenced at x; contents of memory location whose address is x

<x:y> - a modifier which delimits an extent from bit position x to bit position y inclusive

<x1,x2,...,xn> - a modifier which enumerates bits x1,x2,...,xn

{ } - arithmetic parentheses used to indicate precedence

AND - logical AND

OR - logical OR

XOR - logical XOR

NOT - logical (ones) complement

LSS - less than signed

LSSU - less than unsigned

LEQ - less than or equal signed

LEQU - less than or equal unsigned

EQL - equal signed

EQLU - equal unsigned

NEQ - not equal signed

NEQU - not equal unsigned

GEQ - greater than or equal signed

GEQU - greater than or equal unsigned

GTR - greater than signed

GTRU - greater than unsigned

SEXT(x) - x is sign extended to size of operand needed

ZEXT(x) - x is zero extended to size of operand needed

REM(x,y) - remainder of x divided by y, such that x/y and REM(x,y) have the same sign

MINU(x,y) - minimum unsigned of x and y

MAXU(x,y) - maximum unsigned of x and y

The following conventions are used:

1. Other than that caused by ( )+, or -( ), and the advancement of PC, only operands or portions of operands appearing on the left side of assignment statements are affected.

2. No operator precedence is assumed, other than that replacement (<-) has the lowest precedence. Precedence is indicated explicitly by { }.

3. All arithmetic, logical, and relational operators are defined in the context of their operands. For example "+" applied to floating operands means a floating add while "+" applied to byte operands is an integer byte add. Similarily, "LSS" is a floating comparison when applied to floating operands while "LSS" is an integer byte comparison when applied to byte operands.

4. Instruction operands are evaluated according to the operand specifier conventions (See Chapter 3). The order in which operands appear in the instruction description has no effect on the order of evaluation.

5. Condition codes are in general affected on the value of actual stored results, not on "true" results (which might be generated internally to greater precision). Thus, for example, 2 positive integers can be added together and the sum stored, because of overflow, as a negative value. The condition codes will indicate a negative value even though the "true" result is clearly positive.

4.2 INTEGER ARITHMETIC AND LOGICAL INSTRUCTIONS

The following instructions are described in this section.

		Instructions
1.	Add Aligned Word ADAWI add.rw, sum.mw	1
2.	Add 2 Operand ADD{B,W,L}2 add.rx, sum.mx	3
3.	Add 3 Operand ADD{B,W,L}3 add1.rx, add2.rx, sum.wx	3
4.	Add With Carry ADWC add.rl, sum.ml	1
5.	Arithmetic Shift ASH{L,Q} cnt.rb, src.rx, dst.wx	2
6.	Bit Clear 2 Operand BIC{B,W,L}2 mask.rx, dst.mx	3
7.	Bit Clear 3 Operand BIC{B,W,L}3 mask.rx, src.rx, dst.wx	3
8.	Bit Set 2 Operand BIS{B,W,L}2 mask.rx, dst.mx	3
9.	Bit Set 3 Operand BIS{B,W,L}3 mask.rx, src.rx, dst.wx	3
1Ø.	Bit Test BIT{B,W,L} mask.rx, src.rx	3
11.	Clear CLR{B,W,L,Q} dst.wx	4
12.	Compare CMP{B,W,L} src1.rx, src2.rx	3
13.	Convert CVT{B,W,L}{B,W,L} src.rx, dst.wy All pairs except BB,WW,LL.	6
14.	Decrement DEC{B,W,L} dif.mx	3
15.	Divide 2 Operand DIV{B,W,L}2 divr.rx, quo.mx	3

16. Divide 3 Operand 3
DIV{B,W,L}3 divr.rx, divd.rx, quo.wx

17. Extended Divide 1
EDIV divr.rl, divd.rq, quo.wl, rem.wl

18. Extended Multiply 1
EMUL mulr.rl, muld.rl, add.rl, prod.wq

19. Increment 3
INC{B,W,L} sum.mx

20. Move Complemented 3
MCOM{B,W,L} src.rx, dst.wx

21. Move Negated 3
MNEG{B,W,L} src.rx, dst.wx

22. Move 4
MOV{B,W,L,Q} src.rx, dst.wx

23. Move Zero-Extended 3
MOVZ{BW,BL,WL} src.rx, dst.wy

24. Multiply 2 Operand 3
MUL{B,W,L}2 mulr.rx, prod.mx

25. Multiply 3 Operand 3
MUL{B,W,L}3 mulr.rx, muld.rx, prod.wx

26. Push Long 1
PUSHL src.rl, {-(SP).wl}

27. Rotate Long 1
ROTL cnt.rb, src.rl, dst.wl

28. Add Aligned Word 1

29. Subtract With Carry 1
SBWC sub.rl, dif.ml

30. Subtract 2 Operand 3
SUB{B,W,L}2 sub.rx, dif.mx

31. Subtract 3 Operand 3
SUB{B,W,L}3 sub.rx, min.rx, dif.wx

32. Test 3
TST{B,W,L} src.rx

33. Exclusive OR 2 Operand 3
XOR{B,W,L}2 mask.rx, dst.mx

34. Exclusive OR 3 Operand 3
```
XOR{B,W,L}3 mask.rx, src.rx, dst.wx
```

ADAWI Add Aligned Word Interlocked

Format:

```
        opcode add.rw, sum.mw
```

Operation:

```
        tmp <- add;
        {set interlock};
        sum <- sum + tmp;
        {release interlock};
```

Condition Codes:

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

Exceptions:

```
        reserved operand fault
        integer overflow
```

Opcodes:

```
  58    ADAWI  Add Aligned Word Interlocked
```

Description:

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

Notes:

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

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

ADD Add

Format:

```
opcode add.rx, sum.mx           2 operand

opcode add1.rx, add2.rx, sum.wx 3 operand
```

Operation:

```
sum <- sum + add;         !2 operand

sum <- add1 + add2;       !3 operand
```

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

ADWC Add With Carry

Format:

```
        opcode add.rl, sum.ml
```

Operation:

```
        sum <- sum + add + C;
```

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

D8	ADWC	Add With Carry

Description:

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

Notes:

1. On overflow, the sum operand is replaced by the low order bits of the true result.

2. The 2 additions in the operation are performed simultaneously.

ASH Arithmetic Shift

Format:

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

Operation:

```
        dst <- src shifted cnt bits;
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- {integer overflow};
        C <- Ø;
```

Exceptions:

integer overflow

Opcodes:

78	ASHL	Arithmetic Shift Long
79	ASHQ	Arithmetic Shift Quad

Description:

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

Notes:

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

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

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

BIC Bit Clear

Format:

```
        opcode mask.rx, dst.mx          2 operand

        opcode mask.rx, src.rx, dst.wx  3 operand
```

Operation:

```
        dst <- dst AND {NOT mask};      !2 operand

        dst <- src AND {NOT mask};      !3 operand
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

In 2 operand format, the destination operand is ANDed with the ones complement of the mask operand and the destination operand is replaced by the result. In 3 operand format, the source operand is ANDed with the ones complement of the mask operand and the destination operand is replaced by the result.

BIS Bit Set

Format:

```
        opcode mask.rx, dst.mx          2 operand

        opcode mask.rx, src.rx, dst.wx  3 operand
```

Operation:

```
        dst <- dst OR mask;      !2 operand

        dst <- src OR mask;      !3 operand
```

Conditon Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

In 2 operand format, the mask operand is ORed with the destination operand and the destination operand is replaced by the result. In 3 operand format, the mask operand is ORed with the source operand and the destination operand is replaced by the result.

BIT Bit Test

Format:

```
        opcode mask.rx, src.rx
```

Operation:

```
        tmp <- src AND mask;
```

Conditon Codes:

```
        N <- tmp LSS Ø;
        Z <- tmp EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

The mask operand is ANDed with the source operand. Both operands are unaffected. The only action is to affect condition codes.

CLR Clear

Format:

```
opcode dst.wx
```

Operation:

```
dst <- Ø;
```

Condition Codes:

```
N <- Ø;
Z <- 1;
V <- Ø;
C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

The destination operand is replaced by Ø.

Notes:

CLRx dst is equivalent to MOVx S^#Ø, dst, but is 1 byte shorter.

CMP Compare

Format:

```
        opcode src1.rx, src2.rx
```

Operation:

```
        src1 - src2;
```

Condition Codes:

```
        N <- src1 LSS src2;
        Z <- src1 EQL src2;
        V <- 0;
        C <- src1 LSSU src2;
```

Exceptions:

none

Opcodes:

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

Description:

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

CVT Convert

Format:

```
        opcode src.rx, dst.wy
```

Operation:

```
        dst <- conversion of src;
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- {integer overflow};
        C <- Ø;
```

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

DEC Decrement

Format:

```
opcode dif.mx
```

Operation:

```
dif <- dif - 1;
```

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

1. Integer overflow occurs if the largest negative integer is decremented. On overflow, the difference operand is replaced by the largest positive integer.
2. DECx dif is equivalent to SUBx S^#1, dif, but is 1 byte shorter.

DIV Divide

Format:

```
        opcode divr.rx, quo.mx                 2 operand

        opcode divr.rx, divd.rx, quo.wx        3 operand
```

Operation:

```
        quo <- quo / divr;        !2 operand

        quo <- divd / divr;       !3 operand
```

Condition Codes:

```
        N <- quo LSS Ø;
        Z <- quo EQL Ø;
        V <- {integer overflow} OR {divr EQL Ø};
        C <- Ø;
```

Exceptions:

integer overflow
divide by zero

Opcodes:

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

Description:

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

Notes:

1. Division is performed such that the remainder (unless it is zero and which is lost) has the same sign as the dividend, i.e., the result is truncated towards Ø.

2. Integer overflow occurs if and only if the largest negative integer is divided by -1. On overflow, operands are affected as in 3 below.

3. If the divisor operand is Ø, then in 2 operand format the quotient operand is not affected; in 3 operand format the quotient operand is replaced by the dividend operand.

EDIV Extended Divide

Format:

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

Operation:

```
quo <- divd / divr;
rem <- REM(divd, divr);
```

Condition Codes:

```
N <- quo LSS Ø;
Z <- quo EQL Ø;
V <- {integer overflow} OR {divr EQL Ø};
C <- Ø;
```

Exceptions:

integer overflow
divide by zero

Opcodes:

7B	EDIV	Extended Divide

Description:

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

Notes:

1. The division is performed such that the remainder operand (unless it is Ø) has the same sign as the dividend operand.
2. On overflow, the operands are affected as in 3. below.
3. If the divisor operand is Ø, then the quotient operand is replaced by bits 31:Ø of the dividend operand, and the remainder operand is replaced by Ø.

EMUL Extended Multiply

Format:

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

Operation:

```
        prod <- {muld * mulr} + SEXT(add);
```

Condition Codes:

```
        N <- prod LSS Ø;
        Z <- prod EQL Ø;
        V <- Ø;
        C <- Ø;
```

Exceptions:

        none

Opcodes:

7A	EMUL	Extended Multiply

Description:

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

INC Increment

Format:

```
        opcode sum.mx
```

Operation:

```
        sum <- sum + 1;
```

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

2. INCx sum is equivalent to ADDx S^#1, sum, but is 1 byte shorter.

MCOM Move Complemented

Format:

```
        opcode src.rx, dst.wx
```

Operation:

```
        dst <- NOT src;
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

The destination operand is replaced by the ones complement of the source operand.

MNEG Move Negated

Format:

```
        opcode src.rx, dst.wx
```

Operation:

```
        dst <- -src;
```

Condition Codes:

```
        N <- dst LSS 0;
        Z <- dst EQL 0;
        V <- {integer overflow};
        C <- dst NEQ 0;
```

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

MOV Move

Format:

```
        opcode src.rx, dst.wx
```

Operation:

```
        dst <- src;
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

none

Opcodes:

9Ø	MOVB	Move Byte
BØ	MOVW	Move Word
DØ	MOVL	Move Long
7D	MOVQ	Move Quad
7DFD	MOVO	Move Octa

Description:

The destination operand is replaced by the source operand.

MOVZ Move Zero-Extended

Format:

```
        opcode src.rx, dst.wy
```

Operation:

```
        dst <- ZEXT(src);
```

Condition Codes:

```
        N <- Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

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

MUL Multiply

Format:

```
        opcode mulr.rx, prod.mx                 2 operand

        opcode mulr.rx, muld.rx, prod.wx        3 operand
```

Operation:

```
        prod <- prod * mulr;     !2 operand

        prod <- muld * mulr;     !3 operand
```

Condition Codes:

```
        N <- prod LSS 0;
        Z <- prod EQL 0;
        V <- {integer overflow};
        C <- 0;
```

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

PUSHL Push Long

Format:

```
        opcode src.rl
```

Operation:

```
        -(SP) <- src;
```

Condition Codes:

```
        N <- src LSS 0;
        Z <- src EQL 0;
        V <- 0;
        C <- C;
```

Exceptions:

none

Opcodes:

```
  DD    PUSHL  Push Long
```

Description:

The longword source operand is pushed on the stack.

Notes:

PUSHL is equivalent to MOVL src, -(SP), but is 1 byte shorter.

ROTL Rotate Long

Format:

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

Operation:

```
        dst <- src rotated cnt bits;
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

none

Opcodes:

```
  9C    ROTL  Rotate Long
```

Description:

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

SBWC Subtract With Carry

Format:

```
opcode sub.rl, dif.ml
```

Operation:

```
dif <- dif - sub - C;
```

Condition Codes:

```
N <- dif LSS Ø;
Z <- dif EQL Ø;
V <- {integer overflow};
C <- {borrow into most significant bit};
```

Exceptions:

integer overflow

Opcodes:

D9	SBWC	Subtract With Carry

Description:

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

Notes:

1. On overflow, the difference operand is replaced by the low order bits of the true result.

2. The 2 subtractions in the operation are performed simultaneously.

SUB Subtract

Format:

```
        opcode sub.rx, dif.mx            2 operand

        opcode sub.rx, min.rx, dif.wx    3 operand
```

Operation:

```
        dif <- dif - sub;        !2 operand

        dif <- min - sub;        !3 operand
```

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

TST Test

Format:

```
        opcode src.rx
```

Operation:

```
        src - Ø;
```

Condition Codes:

```
        N <- src LSS Ø;
        Z <- src EQL Ø;
        V <- Ø;
        C <- Ø;
```

Exceptions:

none

Opcodes:

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

Description:

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

Notes:

TSTx src is equivalent to CMPx src, S^#Ø, but is 1 byte shorter.

XOR Exclusive OR

Format:

```
        opcode mask.rx, dst.mx          2 operand

        opcode mask.rx, src.rx, dst.wx  3 operand
```

Operation:

```
        dst <- dst XOR mask;     !2 operand

        dst <- src XOR mask;     !3 operand
```

Condition Codes:

```
        N <- dst LSS 0;
        Z <- dst EQL 0;
        V <- 0;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

In 2 operand format, the mask operand is XORed with the destination operand and the destination operand is replaced by the result. In 3 operand format, the mask operand is XORed with the source operand and the destination operand is replaced by the result.

4.3 ADDRESS INSTRUCTIONS

The following instructions are described in this section.

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

MOVA Move Address

Format:

```
        opcode src.ax, dst.wl
```

Operation:

```
        dst <- src;
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

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

Notes:

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

PUSHA Push Address

Format:

```
opcode src.ax
```

Operation:

```
-(SP) <- src;
```

Condition Codes:

```
N <- src LSS Ø;
Z <- src EQL Ø;
V <- Ø;
C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

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

Notes:

1. PUSHAx src is equivalent to MOVAx src, -(SP), but is 1 byte shorter.

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

4.4 VARIABLE LENGTH BIT FIELD INSTRUCTIONS

A variable length bit field is specified by 3 operands:

1. A longword position operand.

2. A byte field size operand which must be in the range 0 through 32 or a reserved operand fault occurs.

3. A base address (relative to which the position is used to locate the bit field). The address is obtained from an operand of address access type. However, unlike other instances of operand specifiers of address access type, register mode may be designated in the operand specifier. In this case the field is contained in the register n designated by the operand specifier (or register n+1 concatenated with register n). (See Chapter 2) If the field is contained in a register and size is not zero, the position operand must have a value in the range 0 through 31 or a reserved operand fault occurs.

In order to simplify the description of the variable bit field instructions, a macro FIELD(pos, size, address) is introduced with the following expansion (if size NEQ 0):

```
FIELD(pos, size, address)

=(address + SEXT(pos<31:3>))<{size - 1} + pos<2:0>:pos<2:0>>

        !if address not specified by register mode

= {R[n+1]'Rn}<{size - 1} + pos:pos>


        !if address specified by register mode and pos + size
        !GTRU 32

= Rn<{size - 1} + pos:pos>

        !if address specified by register mode and pos + size
        !LEQU 32
```

The number of bytes referenced by the contents ( ) operator above is:

```
        1 +  {{{size - 1} + pos<2:0>} / 8}
```

Zero bytes are referenced if the field size is 0.

The following instructions are described in this section.

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

The following variable bit field instructions are described in the section on Control Instructions.

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

CMP Compare Field

Format:

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

Operation:

```
        tmp <- if size NEQU Ø then SEXT(FIELD (pos,
                size, base)) else Ø;     !CMPV
        tmp - src;

        tmp <- if size NEQU Ø then ZEXT(FIELD (pos,
                size, base)) else Ø;      !CMPZV
        tmp - src;
```

Condition Codes:

```
        N <- tmp LSS src;
        Z <- tmp EQL src;
        V <- Ø;
        C <- tmp LSSU src;
```

Exceptions:

reserved operand

Opcodes:

EC	CMPV	Compare Field
ED	CMPZV	Compare Zero-Extended Field

Description:

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

Notes:

1. A reserved operand fault occurs if:

    1. size GTRU 32.

    2. pos GTRU 31, size NEQ Ø, and the field is contained in the registers.

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

EXT Extract Field

Format:

```
        opcode pos.rl, size.rb, base.vb, dst.wl
```

Operation:

```
        dst <- if size NEQU Ø then SEXT(FIELD(pos, size, base))
                         else Ø;          !EXTV

        dst <- if size NEQU Ø then ZEXT(FIELD(pos, size, base))
                         else Ø;          !EXTZV
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

reserved operand

Opcodes:

EE	EXTV	Extract Field
EF	EXTZV	Extract Zero-Extended Field

Description:

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

Notes:

1. A reserved operand fault occurs if:
    1. size GTRU 32.
    2. pos GTRU 31, size NEQ Ø, and the field is contained in the registers.

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

FF Find First

Format:

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

Operation:

```
        state = if {FFS} then 1 else Ø;
        if size NEQU Ø then
                begin
                tmp1 <- FIELD(startpos, size, base);
                tmp2 <- Ø;
                while {tmp1<tmp2> NEQ state} AND
                        {tmp2 LEQU {size - 1}} do
                        tmp2 <- tmp2 + 1;
                findpos <- startpos + tmp2;
                end
        else
                findpos <- startpos;
```

Condition Codes:

```
        N <- Ø;
        Z <- {bit not found};
        V <- Ø;
        C <- Ø;
```

Exceptions:

reserved operand

Opcodes:

EB	FFC	Find First Clear
EA	FFS	Find First Set

Description:

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

Notes:

1. A reserved operand fault occurs if:

    1. size GTRU 32.

    2. startpos GTRU 31, size NEQ 0, and the field is contained in the registers.

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

INSV Insert Field

Format:

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

Operation:

```
        if size NEQU Ø then FIELD(pos, size, base) <-
                src<{size - 1}:Ø>;
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

reserved operand

Opcodes:

FØ	INSV	Insert Field

Description:

The field specified by the position, size, and base operands is replaced by bits size-1:Ø of the source operand. If the size operand is Ø, the only action is to affect the condition codes.

Notes:

1. A reserved operand fault occurs if:
    1. size GTRU 32.
    2. pos GTRU 31, size NEQ Ø, and the field is contained in the registers.
2. On a reserved operand fault, the field is unaffected and the condition codes are UNPREDICTABLE.

4.5 CONTROL INSTRUCTIONS

In most implementations of the VAX-11 architecture, improved execution speed will result if the target of a control instruction is on an aligned longword boundary.

The following instructions are described in this section.

```
                                                          Instructions
                                                          ------------

 1.  Add Compare and Branch                                     7
     ACB{B,W,L,F,D,G,H} limit.rx, add.rx, index.mx, displ.bw
     Compare is LE on positive add, GE on negative
     add.

 2.  Add One and Branch Less Than or Equal                      1
     AOBLEQ limit.rl, index.ml, displ.bb

 3.  Add One and Branch Less Than                               1
     AOBLSS limit.rl, index.ml, displ.bb

 4.  Conditional Branch                                        12
     B{condition} displ.bb
```

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

```
 5.  Branch on Bit                                              2
     BB{S,C} pos.rl, base.vb, displ.bb, {field.rv}

 6.  Branch on Bit (and modify without interlock)               4
     BB{S,C}{S,C} pos.rl, base.vb, displ.bb, {field.mv}

 7.  Branch on Bit (and modify) Interlocked                     2
     BB{SS,CC}I pos.rl, base.vb, displ.bb, {field.mv}

 8.  Branch on Low Bit                                          2
     BLB{S,C} src.rl, displ.bb
```

9.	Branch With {Byte, Word} Displacement BR{B,W} displ.bx	2
10.	Branch to Subroutine With {Byte, Word} Displacement BSB{B,W} displ.bx, {-(SP).wl}	2
11.	Case CASE{B,W,L} selector.rx, base.rx, limit.rx, displ.bw-list	3
12.	Jump JMP dst.ab	1
13.	Jump to Subroutine JSB dst.ab, {-(SP).wl}	1
14.	Return from Subroutine RSB {(SP)+.rl}	1
15.	Subtract One and Branch Greater Than or Equal SOBGEQ index.ml, displ.bb	1
16.	Subtract One and Branch Greater Than SOBGTR index.ml, displ.bb	1

ACB Add Compare and Branch

Format:

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

Operation:

```
index <- index + add;
if {{add GEQ 0} AND {index LEQ limit}} OR
        {{add LSS 0} AND {index GEQ limit}} then
        PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- index LSS 0;
Z <- index EQL 0;
V <- {integer or floating overflow};
C <- C;
```

Exceptions:

integer overflow
floating overflow
floating underflow
reserved operand

Opcodes:

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

Description:

The addend operand is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If the addend operand is positive (or 0) and the comparison is less than or equal or if the addend is negative and the comparison is greater than or equal, the sign-extended branch displacement is added to PC and PC is replaced by the result.

$

Notes:

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

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

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

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

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

6. Except for 5. above, the C-bit is unaffected.

AOBLEQ Add One and Branch Less Than or Equal

Format:

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

Operation:

```
index <- index + 1;
if index LEQ limit then PC <-
        PC + SEXT(displ);
```

Condition Codes:

```
N <- index LSS Ø;
Z <- index EQL Ø;
V <- {integer overflow};
C <- C;
```

Exceptions:

integer overflow

Opcodes:

F3	AOBLEQ	Add One and Branch Less Than or Equal

Description:

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

Notes:

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

2. The C-bit is unaffected.

AOBLSS Add One and Branch Less Than

Format:

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

Operation:

```
        index <- index + 1;
        if index LSS limit then PC <-
                PC + SEXT(displ);
```

Condition Codes:

```
        N <- index LSS 0;
        Z <- index EQL 0;
        V <- {integer overflow};
        C <- C;
```

Exceptions:

integer overflow

Opcodes:

F2	AOBLSS	Add One and Branch Less Than

Description:

One is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If it is less than, the sign-extended branch displacement is added to the PC and PC is replaced by the result.

Notes:

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

2. The C-bit is unaffected.

B Branch on (condition)

Format:

```
opcode displ.bb
```

Operation:

```
if condition then PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes: Condition

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

Description:

The condition codes are tested and if the condition indicated by the instruction is met, the sign-extended branch displacement is added to the PC and PC is replaced by the result.

Notes:

The VAX-11 conditional branch instructions permit considerable flexibility in branching but require care in choosing the correct branch instruction. The conditional branch instructions are best seen as 3 overlapping groups:

1. Overflow and Carry Group

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

   These instructions are typically used to check for overflow (when overflow traps are not enabled), for multiprecision arithmetic, and for other special purposes.

2. Unsigned Group

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

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

3. Signed Group

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

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

BB Branch on Bit

Format:

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

Operation:

```
        teststate = if {BBS} then 1 else 0;
        if FIELD(pos, 1, base) EQL teststate then
                PC <- PC + SEXT(displ);
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

reserved operand

Opcodes:

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

Description:

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

Notes:

1. See Section 4.5 for definition of FIELD.
2. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.
3. On a reserved operand fault, the condition codes are UNPREDICTABLE.

BB Branch on Bit (and modify without interlock)

Format:

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

Operation:

```
        teststate = if {BBSS or BBSC} then 1 else Ø;
        newstate = if {BBSS or BBCS} then 1 else Ø;
        tmp <- FIELD(pos, 1, base);
        FIELD(pos, 1, base) <- newstate;
        if tmp EQL teststate then
                PC <- PC + SEXT(displ);
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

reserved operand

Opcodes:

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

Description:

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

Notes:

1. See Section 4.5 for definition of FIELD.

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

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

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

BB Branch on Bit Interlocked

Format:

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

Operation:

```
        teststate = if {BBSSI} then 1 else 0;
        newstate = teststate;
        {set interlock};
        tmp <- FIELD(pos, 1, base);
        FIELD(pos, 1, base) <- newstate;
        {release interlock};
        if tmp EQL teststate then
                PC <- PC + SEXT(displ);
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

reserved operand

Opcodes:

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

Description:

The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the PC and PC is replaced by the result. Regardless of whether the branch is effected or not, the tested bit is put in the new state as indicated by the instruction. If the bit is contained in memory, the reading of the state of the bit and the setting of it to the new state is an interlocked operation. No other processor or I/O device can do an interlocked access on the bit during the interlocked operation.

Notes:

1. See Section 4.5 for definition of FIELD
2. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in registers.

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

4. Except for memory interlocking BBSSI is equivalent to BBSS and BBCCI is equivalent to BBCC.

5. This instruction is designed to modify interlocks with other processors or devices. For example, to implement "busy waiting":

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

BLB Branch on Low Bit

Format:

```
        opcode src.rl, displ.bb
```

Operation:

```
        teststate = if {BLBS} then 1 else Ø;
        if src<Ø> EQL teststate then
                PC <- PC + SEXT(displ);
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

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

BR Branch

Format:

```
opcode displ.bx
```

Operation:

```
PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

The sign-extended branch displacement is added to PC and PC is replaced by the result.

BSB Branch To Subroutine

Format:

```
        opcode displ.bx
```

Operation:

```
        -(SP) <- PC;
        PC <- PC + SEXT(displ);
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

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

CASE Case

Format:

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

Operation:

```
        tmp <- selector - base;
        PC <- PC + if tmp LEQU limit then
                SEXT(displ[tmp]) else {2 + 2 * ZEXT(limit)};
```

Condition Codes:

```
        N <- tmp LSS limit;
        Z <- tmp EQL limit;
        V <- 0;
        C <- tmp LSSU limit;
```

Exceptions:

none

Opcodes:

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

Description:

The base operand is subtracted from the selector operand and a temporary is replaced by the result. The temporary is compared with the limit operand and if it is less than or equal unsigned, a branch displacement selected by the temporary value is added to PC and PC is replaced by the result. Otherwise, 2 times the sum of the limit operand and 1 is added to PC and PC is replaced by the result. This causes PC to be moved past the array of branch displacements. Regardless of the branch taken, the condition codes are affected by the comparison of the temporary operand with the limit operand.

Notes:

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

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

JMP Jump

Format:

```
        opcode dst.ab
```

Operation:

```
        PC <- dst;
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

```
        none
```

Opcodes:

```
  17    JMP     Jump
```

Description:

PC is replaced by the destination operand.

JSB Jump to Subroutine

Format:

```
        opcode dst.ab
```

Operation:

```
        -(SP) <- PC;
        PC <- dst;
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

none

Opcodes:

16	JSB	Jump to Subroutine

Description:

PC is pushed on the stack as a longword. PC is replaced by the destination operand.

Notes:

Since the operand specifier conventions cause the evaluation of the destination operand before saving PC, JSB can be used for coroutine calls with the stack used for linkage. The form of such a call is JSB @(SP)+.

RSB Return from Subroutine

Format:

```
        opcode
```

Operation:

```
        PC <- (SP)+;
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

```
        none
```

Opcodes:

Ø5	RSB	Return From Subroutine

Description:

PC is replaced by a longword popped from the stack.

Notes:

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

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

SOBGEQ Subtract One and Branch Greater Than or Equal

Format:

```
opcode index.ml, displ.bb
```

Operation:

```
index <- index - 1;
if index GEQ 0 then PC <-
        PC + SEXT(displ);
```

Condition Codes:

```
N <- index LSS 0;
Z <- index EQL 0;
V <- {integer overflow};
C <- C;
```

Exceptions:

integer overflow

Opcodes:

```
F4    SOBGEQ  Subtract One and Branch Greater Than or Equal
```

Description:

One is subtracted from the index operand and the index operand is replaced by the result. If the index operand is greater than or equal to 0, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes:

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

2. The C-bit is unaffected.

SOBGTR Subtract One and Branch Greater Than

Format:

```
        opcode index.ml, displ.bb
```

Operation:

```
        index <- index - 1;
        if index GTR Ø then PC <-
                PC + SEXT(displ);
```

Condition Codes:

```
        N <- index LSS Ø;
        Z <- index EQL Ø;
        V <- {integer overflow};
        C <- C;
```

Exceptions:

```
        integer overflow
```

Opcodes:

```
  F5    SOBGTR  Subtract One and Branch Greater Than
```

Description:

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

Notes:

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

2. The C-bit is unaffected.

4.6 PROCEDURE CALL INSTRUCTIONS

Three instructions are used to implement a standard procedure calling interface. Two instructions implement the CALL to the procedure; the third implements the matching RETURN. Refer to the VAX/VMS Run Time Library Reference Manual for the procedure calling standard. The CALLG instruction calls a procedure with the argument list actuals in an arbitrary location. The CALLS instruction calls a procedure with the argument list actuals on the stack. Upon return after a CALLS this list is automatically removed from the stack. Both call instructions specify the address of the entry point of the procedure being called. The entry point is assumed to consist of a word termed the entry mask followed by the procedure's instructions. The procedure terminates by executing a RET instruction.

The entry mask specifies the subprocedure's register use and overflow enables:

```
 1 1 1 1 1
 5 4 3 2 1                       Ø
+-+-+---+------------------------+
|D|I|MBZ|       REGISTERS        |
|V|V|   |                        |
+-+-+---+------------------------+
```

On CALL the stack is aligned to a longword boundary and the trap enables in the PSW are set to a known state to ensure consistent behavior of the called procedure. Integer overflow enable and decimal overflow enable are affected according to bits 14 and 15 of the entry mask respectively. Floating underflow enable is cleared. The registers R11 through RØ specified by bits 11 through Ø respectively are saved on the stack and are restored by the RET instruction. In addition, PC, SP, FP, and AP are always preserved by the CALL instructions and restored by the RET instruction.

All external procedure CALLs generated by standard DIGITAL language processors, and all inter-module CALLs to major VAX-11 software subsystems comply with the procedure calling software standard (see VAX/VMS Run Time Library Reference Manual, Appendix C). The procedure calling standard requires that all registers in the range R2 through R11 used in the procedure must appear in the mask. RØ and R1 are not preserved by any called procedure that complies with the procedure calling standard.

In order to preserve the state, the CALL instructions form a structure on the stack termed a call frame or stack frame. This contains the saved registers, the saved PSW, the register save mask, and several control bits. The frame also includes a longword which the CALL instructions clear; this is used to implement the condition handling facility. Refer to Appendix D. At the end of execution of the CALL instruction, FP contains the address of the stack frame. The RET instruction uses the contents of FP to find the stack frame and restore state. The condition handling facility assumes that FP always points to the stack frame. The stack frame has the following format:

```
+-----------------------------------------------------------------+
|              condition handler (initially Ø)                   | :(FP)
+---+-+-+---------------------+----------------------+----------+
|SPA|S|Ø|        mask<11:Ø>   |  saved PSW<15:5>     |    Ø     |
+---+-+-+---------------------+----------------------+----------+
|                          saved AP                             |
+---------------------------------------------------------------+
|                          saved FP                             |
+---------------------------------------------------------------+
|                          saved PC                             |
+---------------------------------------------------------------+
|                          saved RØ (...)                       |
+---------------------------------------------------------------+
        .                                                 .
        .                                                 .
        .                                                 .
+---------------------------------------------------------------+
|                          saved R11 (...)                      |
+---------------------------------------------------------------+
    (Ø to 3 bytes specified by SPA, Stack Pointer Alignment)

        S = set if CALLS; clear if CALLG.
```

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

The contents of the frame PSW<3:Ø> at the time RET is executed will become the condition codes resulting from the execution of the procedure. Similarly, the content of the frame PSW<4> at the time the RET is executed will become the PSW<T> bit.
The following instructions are described in this section.

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

CALLG Call Procedure With General Argument List

Format:

```
        opcode arglist.ab, dst.ab
```

Operation:

```
        {align stack};
        {create stack frame};
        {set arithmetic exception enables};
        {set new values of AP,FP,PC};
```

Condition Codes:

```
        N <- Ø;
        Z <- Ø;
        V <- Ø;
        C <- Ø;
```

Exceptions:

        reserved operand

Opcodes:

```
  FA    CALLG   Call Procedure with General Argument List
```

Description:

SP is saved in a temporary and then bits 1:Ø are replaced by Ø so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to Ø and the contents of registers whose number corresponds to set bits in the mask are pushed on the stack as longwords. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:3Ø, a Ø in bit 29 and bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:Ø with T cleared is pushed on the stack. A longword Ø is pushed on the stack. FP is replaced by SP. AP is replaced by the arglist operand. The trap enables in the PSW are set to a known state. Integer overflow, and decimal overflow are affected according to bits 14 and 15 of the entry mask respectively; floating underflow is cleared. T-bit is unaffected. PC is replaced by the sum of destination operand plus 2 which transfers control to the called procedure at the byte beyond the entry mask.

```
+----------------------------------------------------------------+ :(SP)
|                                                                | :(FP)
|                      stack                                     |
|                                                                |
|                              frame                             |
|                                                                |
+----------------------------------------------------------------+

                 (Ø to 3 bytes specified by SPA)
```

Notes:

1. If bits 13:12 of the entry mask are not Ø, a reserved operand fault occurs.
2. On a reserved operand fault, condition codes are UNPREDICTABLE.
3. The procedure calling standard and the condition handling facility require the following register saving conventions. RØ and R1 are always available for function return values and are never saved in the entry mask. All registers R2 through R11 which are modified in the called procedure must be preserved in the mask. Refer to VAX/VMS Run Time Library Reference Manual, Appendix C.

CALLS Call Procedure with Stack Argument List

Format:

```
        opcode numarg.rl, dst.ab
```

Operation:

```
        {push arg count};
        {align stack};
        {create stack frame};
        {set arithmetic exception enables};
        {set new values of AP,FP,PC};
```

Condition Codes:

```
        N <- Ø;
        Z <- Ø;
        V <- Ø;
        C <- Ø;
```

Exceptions:

reserved operand

Opcodes:

```
  FB    CALLS   Call Procedure With Stack Argument List
```

Description:

The numarg operand is pushed on the stack as a longword (byte Ø contains the number of arguments, high order 24 bits are used by DIGITAL software). SP is saved in a temporary and then bits 1:Ø of SP are replaced by Ø so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit Ø and the contents of registers whose number corresponds to set bits in the mask are pushed on the stack. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:3Ø, a 1 in bit 29, a Ø in bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:Ø with T cleared is pushed on the stack. A longword Ø is pushed on the stack. FP is replaced by SP. AP is set to the value of the stack pointer after the numarg operand was pushed on the stack. The trap enables in the PSW are set to a known state. Integer overflow, and decimal overflow, are affected according to bits 14 and 15 of the entry mask, respectively, floating underflow is cleared. T-bit is unaffected.PC is replaced by the sum of destination operand plus 2 which transfers control to the called procedure at the byte beyond the entry mask. The appearance of the stack after CALLS is executed is:

```
+-------------------------------------------------------------+ :(SP)
|                                                             | :(FP)
|                         stack                               |
|                                                             |
|                                                             |
|                              frame                          |
|                                                             |
+-------------------------------------------------------------+

               (Ø to 3 bytes specified by SPA)

+---------------------------------------------+---------------+
|                                             |       N       | :(AP)
+---------------------------------------------+---------------+
         .
         .        N longwords of argument list       .
         .                                           .
+-------------------------------------------------------------+
```

Notes:

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

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

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

4. The procedure calling standard and the condition handling facility require the following register saving conventions. RØ and R1 are always available for function return values and are never saved in the entry mask. All registers R2 through R11 which are modified in the called procedure must be preserved in the entry mask. Refer to VAX/VMS Run Time Library Reference Manual, Appendix C.

RET Return from Procedure

Format:

```
        opcode
```

Operation:

```
        {restore SP from FP};
        {restore registers};
        {drop stack alignment};
        {if CALLS then remove arglist};
        {restore PSW};
```

Condition Codes:

```
        N <- tmpl<3>;
        Z <- tmpl<2>;
        V <- tmpl<1>;
        C <- tmpl<Ø>;
```

Exceptions:

        reserved operand

Opcodes:

Ø4	RET	Return from Procedure

Description:

SP is replaced by FP plus 4. A longword containing stack alignment bits in bits 31:3Ø, a CALLS/CALLG flag in bit 29, the low 12 bits of the procedure entry mask in bits 27:16, and a saved PSW in bits 15:Ø is popped from the stack and saved in a temporary. PC, FP, and AP are replaced by longwords popped from the stack. A register restore mask is formed from bits 27:16 of the temporary. Scanning from bit Ø to bit 11 of the restore mask, the contents of registers whose number is indicated by set bits in the mask are replaced by longwords popped from the stack. SP is incremented by 31:3Ø of the temporary. PSW is replaced by bits 15:Ø of the temporary. If bit 29 in the temporary is 1 (indicating that the procedure was called by CALLS), a longword containing the number of arguments is popped from the stack. Four times the unsigned value of the low byte of this longword is added to SP and SP is replaced by the result.

Notes:

1. A reserved operand fault occurs if tmpl<15:8> NEQ Ø.

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

3. The value of tmpl<28> is ignored.

4. The procedure calling standard and condition handling facility assume that procedures which return a function value or a status code do so in RØ or RØ and R1. Refer to VAX/VMS Run Time Library Reference Manual, Appendix C.

4.7 MISCELLANEOUS INSTRUCTIONS

The following instructions are described in this section.

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

BICPSW Bit Clear PSW

Format:

```
        opcode mask.rw
```

Operation:

```
        PSW <- PSW AND {NOT mask};
```

Condition Codes:

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

Exceptions:

```
        reserved operand
```

Opcodes:

```
  B9     BICPSW  Bit Clear PSW
```

Description:

PSW is ANDed with the ones complement of the mask operand and PSW is replaced by the result.

Notes:

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

BISPSW Bit Set PSW

Format:

```
        opcode mask.rw
```

Operation:

```
        PSW <- PSW OR mask;
```

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

B8	BISPSW	Bit Set PSW

Description:

PSW is ORed with the mask operand and PSW is replaced by the result.

Notes:

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

BPT Breakpoint Fault

Format:

```
        opcode
```

Operation:

```
        PSL<TP> <- Ø;
        {breakpoint fault};     !push current PSL on stack
```

Condition Codes:

```
        N <- Ø; !condition codes cleared after BPT fault
        Z <- Ø;
        V <- Ø;
        C <- Ø;
```

Exceptions:

```
        none
```

Opcodes:

```
  Ø3    BPT     Breakpoint Fault
```

Description:

In order to understand the operation of this instruction, it is necessary to read Chapter 6. This instruction is used, together with the T-bit, to implement debugging facilities.

HALT Halt

Format:

```
        opcode
```

Operation:

```
        If PSL<current_mode> NEQU kernel then
                {privileged instruction fault}
                else
                {halt the processor};
```

Condition Codes:

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

        N <- N; !If processor halt
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

privileged instruction

Opcodes:

```
  00    HALT    Halt
```

Description:

In order to understand the operation of this instruction it is necessary to read Chapter 6. If the process is running in kernel mode, the processor is halted. Otherwise, a privileged instruction fault occurs.

Notes:

This opcode is 0 to trap many branches to data.

INDEX Compute Index

Format:

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

Operation:

```
        indexout <- {indexin + subscript} *size;
        if {subscript LSS low} or {subscript GTR high}
        then {subscript range trap};
```

Condition Codes:

```
        N <- indexout LSS 0;
        Z <- indexout EQL 0;
        V <- 0;
        C <- 0;
```

Exceptions:

        subscript range

Opcodes:

```
  0A    INDEX   index
```

Description:

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

Notes:

1. No arithmetic exception other than subscript range can result from this instruction. Thus no indication is given if overflow occurs in either the add or multiply steps. If overflow occurs on the add step the sum is the low order 32 bits of the true result. If overflow occurs on the multiply step, the indexout operand is replaced by the low order 32 bits of the true product of the sum and the subscript operand. In the normal use of this instruction, overflow cannot occur without a subscript range trap occurring.

2. The index instruction is useful in index calculations for arrays of the fixed length data types (integer and floating) and for index calculations for arrays of bit fields, character strings, and decimal strings. The indexin operand permits cascading INDEX instructions for multidimensional arrays. For

one-dimensional bit field arrays it also permits introduction of the constant portion of an index calculation which is not readily absorbed by address arithmetic. The following notes will show some of the uses of INDEX.

3. The COBOL statements:

```
01  A-ARRAY.

    02  A PIC X(10) OCCURS 15 TIMES.

01  B PIC X(10).

    MOVE A(I) TO B.
```

could compile to:

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

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

4. The PL/1 statements:

```
DCL A(-3:10) BIT (5);

A(I) = 1;
```

could compile to:

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

    INSV  #1, R0, #5, A; assumes A byte aligned
```

5. The FORTRAN statements:

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

A(I,J) = 1
```

could compile to:

```
    INDEX  J, #L2, #U2, #M1, #0, R0; M1=U1-L1+1

    INDEX  I, #L1, #U1, #1, R0, R0;

    MOVL   #1, A-a[R0]; a = {{L2*M1} + L1} *4
```

MOVPSL Move from PSL

Format:

```
        opcode dst.wl
```

Operation:

```
        dst <- PSL;
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

none

Opcodes:

```
  DC    MOVPSL  Move from PSL
```

Description:

The destination operand is replaced by PSL (See Chapter 6).

NOP No Operation

Format:

```
        opcode
```

Operation:

```
        none
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

```
        none
```

Opcodes:

```
  Ø1    NOP     No Operation
```

Description:

No operation is performed.

POPR Pop Registers

Format:

```
        opcode mask.rw
```

Operation:

```
        for tmp <- Ø step 1 until 14 do
        if mask<tmp> EQL 1 then R[tmp] <- (SP)+;
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

none

Opcodes:

BA	POPR	Pop Registers

Description:

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

PUSHR Push Registers

Format:

```
        opcode mask.rw
```

Operation:

```
        for tmp <- 14 step -1 until 0 do
        if mask<tmp> EQL 1 then -(SP) <- R[tmp];
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

none

Opcodes:

BB	PUSHR	Push Registers

Description:

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

Notes:

The order of pushing is specified so that the contents of higher numbered registers are stored at higher memory addresses. This results in, say, a double floating datum stored in adjacent registers being stored by PUSHR in memory in the correct order.

XFC Extended Function Call

Format:

```
        opcode
```

Operation:

```
        {XFC fault};
```

Condition Codes:

```
        N <- Ø;
        Z <- Ø;
        V <- Ø;
        C <- Ø;
```

Exceptions:

none

Opcodes:

FC	XFC	Extended Function Call

Description:

In order to understand the operation of this instruction, it is necessary to read Chapter 6. This instruction provides for customer defined extensions to the instruction set.

4.8 QUEUE INSTRUCTIONS

A queue is a circular, doubly linked list. A queue entry is specified by its address. Each queue entry is linked to the next via a pair of longwords. The first longword is the forward link : it specifies the location of the succeeding entry. The second longword is the backward link : it specifies the location of the preceeding entry. The VAX-11 supports two distinct types of links : absolute, and self-relative. An absolute link contains the absolute address of the entry that it points to. A self-relative link contains a displacement from the present queue entry. A queue is classified by the type of link it uses.

4.8.1 Absolute Queues

Absolute queues use absolute addresses as links. Queue entries are linked by a pair of longwords.

The first (lowest addressed) longword is the forward link: the address of the succeeding queue entry. The second (highest addressed) longword is the backward link: the address of the preceding queue entry. A queue is specified by a queue header which is identical to a pair of queue linkage longwords. The forward link of the header is the address of the entry termed the head of the queue. The backward link of the header is the address of the entry termed the tail of the queue. The forward link of the tail points to the header.

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

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

```
 3
 1                                                         0
+-----------------------------------------------------------+
|                 H                                         | :H
+-----------------------------------------------------------+
|                 H                                         | :H+4
+-----------------------------------------------------------+
 3                                                         0
 1
```

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

```
 3
 1                                                              Ø
+---------------------------------------------------------------+
|                  B                                            | :H
+---------------------------------------------------------------+
|                  B                                            | :H+4
+---------------------------------------------------------------+
 3                                                              Ø
 1


 3
 1                                                              Ø
+---------------------------------------------------------------+
|                  H                                            | :B
+---------------------------------------------------------------+
|                  H                                            | :B+4
+---------------------------------------------------------------+
 3                                                              Ø
 1
```

If an entry at address A is inserted at the head of the queue, the queue is as shown below:

```
 3
 1                                                              0
+---------------------------------------------------------------+
|                           A                                   | :H
+---------------------------------------------------------------+
|                           B                                   | :H+4
+---------------------------------------------------------------+
 3                                                              0
 1


 3
 1                                                              0
+---------------------------------------------------------------+
|                           B                                   | :A
+---------------------------------------------------------------+
|                           H                                   | :A+4
+---------------------------------------------------------------+
 3                                                              0
 1


 3
 1                                                              0
+---------------------------------------------------------------+
|                           H                                   | :B
+---------------------------------------------------------------+
|                           A                                   | :B+4
+---------------------------------------------------------------+
 3                                                              0
 1
```

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

```
 3
 1                                                             0
+-------------------------------------------------------------+
|                  A                                          | :H
+-------------------------------------------------------------+
|                  C                                          | :H+4
+-------------------------------------------------------------+
 3                                                             0
 1


 3
 1                                                             0
+-------------------------------------------------------------+
|                  B                                          | :A
+-------------------------------------------------------------+
|                  H                                          | :A+4
+-------------------------------------------------------------+
 3                                                             0
 1


 3
 1                                                             0
+-------------------------------------------------------------+
|                  C                                          | :B
+-------------------------------------------------------------+
|                  A                                          | :B+4
+-------------------------------------------------------------+
 3                                                             0
 1


 3
 1                                                             0
+-------------------------------------------------------------+
|                  H                                          | :C
+-------------------------------------------------------------+
|                  B                                          | :C+4
+-------------------------------------------------------------+
 3                                                             0
 1
```

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

If more than 1 process can perform operations on a queue simultaneously, insertions and removals should only be done at the head or tail of the queue. If only 1 process (or 1 process at a time) can perform operations on a queue, insertions and removals can be made at other than the head or tail of the queue. In the example above with the queue containing entries A,B, and C, the entry at address B can be removed giving:

```
 3
 1                                                              0
+-------------------------------------------------------------+
|                      A                                      | :H
+-------------------------------------------------------------+
|                      C                                      | :H+4
+-------------------------------------------------------------+
 3                                                              0
 1


 3
 1                                                              0
+-------------------------------------------------------------+
|                      C                                      | :A
+-------------------------------------------------------------+
|                      H                                      | :A+4
+-------------------------------------------------------------+
 3                                                              0
 1


 3
 1                                                              0
+-------------------------------------------------------------+
|                      H                                      | :C
+-------------------------------------------------------------+
|                      A                                      | :C+4
+-------------------------------------------------------------+
 3                                                              0
 1
```

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

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

4.8.2 Self-relative Queues

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

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

```
 3
 1                                                             Ø
+---------------------------------------------------------------+
|                           Ø                                   | :H
+---------------------------------------------------------------+
|                           Ø                                   | :H+4
+---------------------------------------------------------------+
 3                                                             Ø
 1
```

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

```
 3
 1                                                             Ø
+---------------------------------------------------------------+
|                         B - H                                 | :H
+---------------------------------------------------------------+
|                         B - H                                 | :H+4
+---------------------------------------------------------------+
 3                                                             Ø
 1
```

```
 3
 1                                                             Ø
+---------------------------------------------------------------+
|                         H - B                                 | :B
+---------------------------------------------------------------+
|                         H - B                                 | :B+4
+---------------------------------------------------------------+
 3                                                             Ø
 1
```

If an entry at address A is inserted at the head of the queue, the queue is as shown below:

```
 3
 1                                                           0
+-----------------------------------------------------------+
|                    A - H                                  | :H
+-----------------------------------------------------------+
|                    B - H                                  | :H+4
+-----------------------------------------------------------+
 3                                                           0
 1


 3
 1                                                           0
+-----------------------------------------------------------+
|                    B - A                                  | :A
+-----------------------------------------------------------+
|                    H - A                                  | :A+4
+-----------------------------------------------------------+
 3                                                           0
 1


 3
 1                                                           0
+-----------------------------------------------------------+
|                    H - B                                  | :B
+-----------------------------------------------------------+
|                    A - B                                  | :B+4
+-----------------------------------------------------------+
 3                                                           0
 1
```

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

```
 3
 1                                                           Ø
+------------------------------------------------------------+
|                    A - H                                   | :H
+------------------------------------------------------------+
|                    C - H                                   | :H+4
+------------------------------------------------------------+
 3                                                           Ø
 1


 3
 1                                                           Ø
+------------------------------------------------------------+
|                    B - A                                   | :A
+------------------------------------------------------------+
|                    H - A                                   | :A+4
+------------------------------------------------------------+
 3                                                           Ø
 1


 3
 1                                                           Ø
+------------------------------------------------------------+
|                    C - B                                   | :B
+------------------------------------------------------------+
|                    A - B                                   | :B+4
+------------------------------------------------------------+
 3                                                           Ø
 1


 3
 1                                                           Ø
+------------------------------------------------------------+
|                    H - C                                   | :C
+------------------------------------------------------------+
|                    B - C                                   | :C+4
+------------------------------------------------------------+
 3                                                           Ø
 1
```

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

Four operations can be performed on self-relative queues : insert at head, insert at tail, remove from head, and remove from tail. Furthermore, these operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared list without additional synchronization. Queue entries must be quadword aligned. Hardware supported interlocked memory access mechanism is used to read the queue header. Bit Ø of the queue header is used as a secondary interlock and is set when the queue is being accessed. If an interlocked queue instruction encounters the secondary interlock set, it terminates after setting the condition codes to indicate failure to gain access to the queue. If the secondary interlock bit is not set, then the interlocked queue instruction sets it during its operation and clears it at instruction completion. This prevents other interlocked queue instructions from operating on the same queue.

4.8.3 Instruction Descriptions

The following instructions are described in this section.

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

INSQHI Insert Entry into Queue at Head, Interlocked

Format:

```
        opcode  entry.ab, header.aq
```

Operation:

```
        tmpl <- (header){interlocked};  !acquire hardware interlock
                                        !must  have  write  access  to
header

                                        !header must be quadword aligned
                                        !header cannot be equal to entry
                                        !tmpl<2:1> must be zero

        if tmpl<0> EQLU 1 then

_RAINBW::_TTA1:,NUNES       15:29:57.79


                begin
                (header){interlocked}  <-  tmpl;  !release  hardware
interlock
                {set condition codes and terminate instruction};
                end;
        else
                begin
                (header){interlocked}  <-  tmpl  v  1;  !set  secondary
interlock
                                                   !release     hardware
interlock
                If {all memory accesses can be completed} then
                        !check if following addresses can be written
                        !without causing a memory management exception:
                        !       entry
                        !       header + tmpl
                        !Also, check for quadword alignment
                        begin
                        {insert entry into queue};
                        {release secondary interlock};
                        end;
                else
                        begin
                        {release secondary interlock};
                        {backup instruction};
                        {initiate fault};
                        end;
                end;
```

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

5C	INSQHI	Insert Entry into Queue at Head, Interlocked

Description:

The entry specified by the entry operand is inserted into the queue following the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (See Chapters 5 and 6), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

Notes:

1. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (See Chapters 5, 6, and 7).

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

3. To set a software interlock realized with a queue, the following can be used:

```
INSERT:          INSQHI  ...              ;was queue empty?
        BEQL     1$                 ;yes
        BCS      INSERT             ;try inserting again
        CALL     WAIT(...)          ;no, wait

1$:
```

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

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

INSQTI Insert Entry into Queue at Tail, Interlocked

Format:

```
        opcode  entry.ab, header.aq
```

Operation:

```
        tmpl <- (header){interlocked};  !acquire hardware interlock
                                        !must  have  write  access  to
header
                                        !header must be quadword aligned
                                        !header cannot be equal to entry
                                        !tmpl<2:1> must be zero

        if tmpl<0> EQLU 1 then
                begin
                (header){interlocked}  <-  tmpl;   !release  hardware
interlock
                {set condition codes and terminate instruction};
                end;
        else
                begin
                (header){interlocked}  <-  tmpl  v  1;  !set  secondary
interlock
                                                   !release      hardware
interlock
                If {all memory accesses can be completed} then
                        !check if the following addresses can be written
                        !without causing a memory management exception:
                        !       entry
                        !       header + (header + 4)
                        !Also, check for quadword alignment
                        begin
                        {insert entry into queue};
                        {release secondary interlock};
                        end;
                else
                        begin
                        {release secondary interlock};
                        {backup instruction};
                        {initiate fault};
                        end;
                end;
```

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

5D	INSQTI	Insert Entry into Queue at Tail, Interlocked

Description:

The entry specified by the entry operand is inserted into the queue preceding the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (See Chapters 5 and 6), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

Notes:

1. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (See Chapters 5, 6, and 7).

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

3. To set a software interlock realized with a queue, the following can be used:

```
INSERT:         INSQHI  ...              ;was queue empty?
        BEQL    1$              ;yes
        BCS INSERT              ;try inserting again
        CALL    WAIT(...)       ;no, wait

1$:
```

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

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

INSQUE Insert Entry in Queue

Format:

```
        opcode  entry.ab, pred.ab
```

Operation:

```
        If {all memory accesses can be completed} then
                begin

                (entry) <- (pred);         !forward link of entry
                (entry + 4) <- pred;       !backward link of entry
                ((pred) + 4) <- entry;     !backward link of successor
                (pred) <- entry;           !forward link of predecessor
                end;
        else
                begin
                {backup instruction};
                {initiate fault};
                end;
```

Condition Codes:

```
        N <- (entry) LSS (entry+4);
        Z <- (entry) EQL (entry+4);       !first entry in queue
        V <- Ø;
        C <- (entry) LSSU (entry+4);
```

Exceptions:

none

Opcodes:

ØE	INSQUE	Insert Entry in Queue

Description:

The entry specified by the entry operand is inserted into the queue following the entry specified by the predecessor operand. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (See Chapters 5 and 6), the queue is left in a consistent state.

Notes:

1. Three types of insertion can be performed by appropriate choice of predecessor operand:

    1. Insert at head

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

    2. Insert at tail

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

    3. Insert after arbitrary predecessor

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

2. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (See Chapters 5, 6, and 7).

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

4. To set a software interlock realized with a queue, the following can be used:

```
        INSQUE  ...              ;was queue empty?
        BEQL    1$               ;yes
        CALL    WAIT(...)        ;no, wait

1$:
```

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

REMQHI Remove Entry from Queue at Head, Interlocked

Format:

```
        opcode  header.aq, addr.wl
```

Operation:

```
        tmpl <- (header){interlocked};  !acquire hardware interlock
                                        !must have write access to
header
                                        !header must be quadword aligned
                                        !header cannot equal address of
addr
                                        !tmpl<2:1> must be zero
        if tmpl<Ø> EQLU 1 then
                begin
                (header){interlocked} <- tmpl;   !release hardware
interlock
                {set condition codes and terminate instruction};
                end;
        else
                begin
                (header){interlocked} <- tmpl v 1; !set secondary
interlock
                                                   !release      hardware
interlock
                If {all memory accesses can be completed} then
                        !check if the following can be done without
                        !causing a memory management exception:
                        !write addr operand
                        !read contents of header + tmpl {if tmpl NEQU Ø}
                        !write into header + tmpl + (header + tmpl) {if
                        !                               tmpl NEQU Ø}
                        !Also, check for quadword alignment
                        begin
                        {remove entry from queue};
                        {release secondary interlock};
                        end;
                else
                        begin
                        {release secondary interlock};
                        {backup instruction};
                        {initiate fault};
                        end;
                end;
```

Condition Codes:

```
        if {removal succeeded} then
                begin
                N <- 0;
                Z <- (header) EQL 0;     !queue empty
                V <- tmpl EQL 0;         !no entry to remove
                C <- 0;
                end;
        else
                begin
                N <- 0;
                Z <- 0;
                V <- 1;              !did not remove anything
                C <- 1;              !secondary interlock failed
                end;
```

Exceptions:

reserved operand

Opcodes:

5E	REMQHI	Remove Entry from Queue at Head, Interlocked

Description:

The queue entry following the header is removed from the queue. The address operand is replaced by the address of the entry removed. If no entry was removed from the queue (because either there was nothing to remove or secondary interlock failed), the condition code V bit is set; otherwise it is cleared. If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (See Chapters 5 and 6), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes:

1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (See Chapters 5, 6, and 7).

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

3. To release a software interlock realized with a queue, the following can be used:

```
1$:     REMQHI  ...                 ;removed last?
        BEQL    2$                  ;yes
        BCS     1$                  ;try removing again
        CALL    ACTIVATE(...)       ;Activate other waiters

2$:
```

4. To remove entries until the queue is empty, the following can be used:

```
1$:     REMQHI  ...                 ;anything removed?
        BVS     2$                  ;no
          .
        process removed entry
          .
        BR      1$                  ;
          .
2$:     BCS     1$                  ;try removing again
        queue empty
```

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

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

REMQTI Remove Entry from Queue at Tail, Interlocked

Format:

```
        opcode  header.aq, addr.wl
```

Operation:

```
        tmpl <- (header){interlocked};  !acquire hardware interlock
                                        !must  have  write  access  to
header
                                        !header must be quadword aligned
                                        !header cannot equal address of
 addr
                                        !tmpl<2:1> must be zero

        if tmpl<Ø> EQLU 1 then
                begin
                (header){interlocked}  <-  tmpl;   !release  hardware
 interlock
                {set condition codes and terminate instruction};
                end;
        else
                begin
                (header){interlocked}  <-  tmpl  v  1;  !set  secondary
 interlock
                                                   !release      hardware
 interlock
                If {all memory accesses can be completed} then
                        !check if the following can be done without
                        !causing a memory management exception :
                        !write addr operand
                        !read contents of header + (header + 4) {if tmpl
                        !                                         NEQU Ø}
                        !write into header + (header + 4)
                        !    + (header + 4 + (header + 4)) {if tmpl NEQU
 Ø}
                        !Also, check for quadword alignment
                        begin
                        {remove entry from queue};
                        {release secondary interlock};
                        end;
                else
                        begin
                        {release secondary interlock};
                        {backup instruction};
                        {initiate fault};
                        end;
                end;
```

Condition Codes:

```
        if {removal succeeded} then
                begin
                N <- 0;
                Z <- (header + 4) EQL 0;        !queue empty
                V <- tmp3 EQL 0                 !no entry to remove
                C <- 0;
                end;
        else
                begin
                N <- 0;
                Z <- 0;
                V <- 1;          !did not remove anything
                C <- 1;          !secondary interlock failed
                end;
```

Exceptions:

reserved operand

Opcodes:

5F   REMQTI Remove Entry from Queue at Tail, Interlocked

Description:

The queue entry preceding the header is removed from the queue. The address operand is replaced by the address of the entry removed. If no entry was removed from the queue (because either there was nothing to remove or secondary interlock failed), the condition code V bit is set; otherwise it is cleared. If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (See Chapters 5 and 6), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes:

1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (See Chapters 5, 6, and 7).

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

3. To release a software interlock realized with a queue, the following can be used:

```
1$:     REMQTI   ...               ;removed last?
        BEQL     2$                ;yes
        BCS      1$                ;try removing again
        CALL     ACTIVATE(...)     ;Activate other waiters

2$:
```

4. To remove entries until the queue is empty, the following can be used:

```
1$:     REMQTI   ...               ;anything removed?
        BVS      2$                ;no
          .
        process removed entry
          .
        BR       1$                ;
          .
2$:     BCS      1$                ;try removing again
        queue empty
```

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

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

REMQUE Remove Entry From Queue

Format:

```
        opcode  entry.ab,addr.wl
```

Operation:

```
        if {all memory acceses can be completed} then
                begin

                ((entry+4)) <- (entry); !forward link of predecessor
                ((entry)+4) <- (entry +4);!backward link of successor
                addr <- entry;
                end;
        else
                begin
                {backup instruction};
                {initiate fault};
                end;
```

Condition Codes:

```
        N <- (entry) LSS (entry+4);

        Z <- (entry) EQL (entry+4);      !queue empty

        V <- entry EQL (entry+4);        !no entry to remove
        C <- (entry) LSSU (entry+4);
```

Exceptions:

none

Opcodes:

ØF	REMQUE	Remove Entry from Queue

Description:

The queue entry specified by the entry operand is removed from the queue. The address operand is replaced by the address of the entry removed. If there was no entry in the queue to be removed, the condition code V bit is set; otherwise it is cleared. If the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (See Chapters 5 and 6), the queue is left in a consistent state.

Notes:

1. Three types of removal can be performed by suitable choice of entry operand:

   1. Remove at head

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

   2. Remove at tail

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

   3. Remove arbitrary entry

      ```
      REMQUE  entry,addr       ;
      ```

2. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (See Chapters 5, 6, and 7).

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

4. To release a software interlock realized with a queue, the followng can be used:

   ```
          REMQUE  ...             ;queue empty?
          BEQL    1$              ;yes
          CALL    ACTIVATE(...)   ;Activate other waiters

   1$:
   ```

5. To remove entries until the queue is empty, the following can be used:

   ```
   1$:    REMQUE  ...             ;anything removed?
          BVS     EMPTY           ;no
            .
            .
            .
          BR      1$              ;
   ```

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

4.9 FLOATING POINT INSTRUCTIONS

The floating point instructions operate on four data types. F_floating and D_floating instructions are standard on all VAX processors. G_floating and H_floating instructions are optional on the VAX-11/78Ø and the VAX-11/75Ø; standard on the VAX-11/73Ø.

In order to be consistent with the floating point instruction set which faults on reserved operands (See Chapter 2), software implemented floating point functions (e.g., the absolute function) should verify that the input operand(s) is (are) not reserved. An easy way to do this is a floating move or test of the input operand(s).

In order to facilitate high speed implementations of the floating point instruction set, certain 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.

Specifically: if within the same instruction the contents of register Rn is used as both a part of a floating point input operand (i.e., a .rf, .rd, .rg, .rh, .mf, .md, .mg, or .mh operand) and as an address in an addressing mode which modifies Rn (i.e., autoincrement, autodecrement, or autoincrement deferred), the value of the floating point operand is UNPREDICTABLE.

4.9.1 Introduction

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

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

where K is an integer and f is a non-negative fraction. For a non-vanishing number, K and f are uniquely determined by imposing the condition

1/2 LEQ f LSS 1.

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

The VAX-11 floating point data formats are derived 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. Sign magnitude notation is used, as follows:

1. Non-zero 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 of course 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 data types, G_floating and H_floating, 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, eight 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 below, a biased EXP of 0 (true exponent of -128), is reserved for floating point zero. Thus, for the 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 eleven bits are reserved for the storage of the exponent in excess 1024 notation. In the H_floating data type fifteen bits are reserved for the storage of the exponent in excess 16384 notation. A biased exponent of 0 is reserved for floating point zero. Thus, exponents are restricted to -1023 to +1023 inclusive (in excess notation, 1 to 2047), and -16383 to +16383 inclusive (in excess notation, 1 to 32767) for the G_floating and H_floating data types respectively.

2. Floating point zero:

Because of the hidden bit, the fractional factor is not available to distinguish between zero and non-zero numbers whose fractional factor is exactly 1/2. Therefore the VAX-11 reserves a sign-exponent field of 0 for this purpose. Any positive floating point number with biased exponent of 0 is treated as if it were an exact 0 by the floating point instruction set. In particular, a floating point operand, whose bits are all 0's, 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 one and a biased exponent of zero. On the VAX-11, 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.

4.9.2 Overview Of The Instruction Set

The VAX-11 has the standard arithmetic operations ADD, SUB, MUL, and DIV implemented for all four floating data types. The results of these operations are always rounded, as described in the section on accuracy. It has, in addition, 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 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 the next section, and exceptions are discussed in Chapter 6.

The VAX-11 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-11 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 the section on accuracy to follow. However, a few may generate rounding error, floating overflow, floating underflow, or induce integer overflow. Details are given in the description of the CVT instructions.

There is a class of move-type instructions which are always exact: MOV, NEG, CLR, CMP, and TST. And, finally, there is the ACB (add, compare and branch) instruction, which is subject to rounding errors, overflow and underflow.

All of the floating point instructions on the VAX-11 fault if a reserved operand is encountered. 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 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 completely discussed in Chapter 6.

4.9.3 Accuracy

General comments on the accuracy of the VAX-11 floating point instruction set are presented here. The descriptions of the individual instructions may include additional details on the accuracy at which they operate.

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

For non-zero 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. We show below that for ADD, SUB, MUL and DIV, an overflow bit, on the left, and two guard bits, on the right, are necessary and sufficient to guarantee return of a rounded result identical to the corresponding infinite precision operation rounded to the specified word length. Thus, with 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 non-zero 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:

1. If the rounding bit is one, the rounded result is the chopped result incremented by an LSB (least significant bit).

2. If the rounding bit is zero, the rounded and chopped results are identical.

All VAX-11 processors implement rounding so as 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 happens, the new rounding bit will be zero, so it can happen only once. The following statements summarize the relations among chopped, rounded and true (infinite precision) results:

1. If a stored result is exact

   rounded value = chopped value = true value.

2. If a stored result is not exact, it's magnitude

   1. is always less than that of the true result for chopping.

2. is always less than that of the true result for rounding if the rounding bit is zero.

3. is greater than that of the true result for rounding if the rounding bit is one.

4.9.4 Instruction Descriptions

The following instructions are described in this section.

		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 1
POLYD arg.rd, degree.rw, tbladdr.ab, {R0-5.wl}

16. Polynomial Evaluation G_floating 1
POLYG arg.rg, degree.rw, tbladdr.ab, {R0-5.wl}

17. Polynomial Evaluation H_floating 1
POLYH arg.rh, degree.rw, tbladdr.ab,
{R0-5.wl,-16(SP):-1(SP).wb}

18. Subtract 2 Operand 4
SUB{F,D,G,H}2 sub.rx, dif.mx

19. Subtract 3 Operand 4
SUB{F,D,G,H}3 sub.rx, min.rx, dif.wx

20. Test 4
TST{F,D,G,H} src.rx

The following floating point instructions are described in the section on Control Instructions.

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

ADD Add

Format:

```
opcode add.rx, sum.mx                     2 operand

opcode add1.rx, add2.rx, sum.wx           3 operand
```

Operation:

```
sum <- sum + add;        !2 operand

sum <- add1 + add2;      !3 operand
```

Condition Codes:

```
N <- sum LSS 0;
Z <- sum EQL 0;
V <- {floating overflow};
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 0 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
```

Operation:

```
        dst <- Ø;
```

Condition Codes:

```
        N <- Ø;
        Z <- 1;
        V <- Ø;
        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 Ø.

Notes:

CLRx dst is equivalent to MOVx #Ø, dst, but is 5 (F_floating) or 9 (D_floating or G_floating) or 17 (H_floating) bytes shorter.

CMP Compare

Format:

```
        opcode src1.rx, src2.rx
```

Operation:

```
        src1 - src2;
```

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.

Notes:

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

CVT Convert

Format:

```
        opcode src.rx, dst.wy
```

Operation:

```
        dst <- conversion of src;
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- {src cannot be represented in dst};
        C <- Ø;
```

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:

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

Notes:

1. Only CVTDF, CVTGF, CVTHF, CVTHD, and CVTHG can result in 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. Zero is stored as the result of floating underflow only if FU is clear. On a floating underflow fault, the destination operand is unaffected. If FU is clear, the destination operand is replaced by 0 and no exception occurs.

DIV Divide

Format:

```
opcode divr.rx, quo.mx           2 operand

opcode divr.rx, divd.rx, quo.wx  3 operand
```

Operation:

```
quo <- quo / divr;        !2 operand

quo <- divd / divr;       !3 operand
```

Condition Codes:

```
N <- quo LSS 0;
Z <- quo EQL 0;
V <- {floating overflow} or {divr EQL 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. Zero is stored as the result of floating underflow only if FU is clear. On a floating underflow fault, the quotient operand is unaffected. If FU is clear, the quotient operand is replaced by Ø 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 3. above.

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
```

Operation:

```
        int <- integer part of muld * {mulr'mulrx};
        fract <- fractional part of muld * {mulr'mulrx};
```

Condition Codes:

```
        N <- fract LSS Ø;
        Z <- fract EQL Ø;
        V <- {integer overflow};
        C <- Ø;
```

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 is such that the 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 in H_floating. Regarding the result 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. The integer and fraction parts are replaced by zero on the occurrence of floating underflow only if FU is clear. On a floating underflow fault, the integer and fraction parts are unaffected. If FU is clear, the integer and fraction parts are replaced by Ø 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
```

Operation:

```
        dst <- -src;
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- Ø;
```

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.
Notes:

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

MOV Move

Format:

```
        opcode src.rx, dst.wx
```

Operation:

```
        dst <- src;
```

Condition Codes:

```
        N <- dst LSS Ø;
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;
```

Exceptions:

reserved operand

Opcodes:

5Ø	MOVF	Move F_floating
7Ø	MOVD	Move D_floating
5ØFD	MOVG	Move G_floating
7ØFD	MOVH	Move H_floating

Description:

The destination operand is replaced by the source operand.

Notes:

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

MUL Multiply

Format:

```
        opcode mulr.rx, prod.mx                  2 operand

        opcode mulr.rx, muld.rx, prod.wx         3 operand
```

Operation:

```
        prod <- prod * mulr;    !2 operand

        prod <- muld * mulr;    !3 operand
```

Condition Codes:

```
        N <- prod LSS Ø;
        Z <- prod EQL Ø;
        V <- {floating overflow};
        C <- Ø;
```

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. Zero is stored as the result of floating underflow only if FU is clear. On a floating underflow fault, the product operand is unaffected. If FU is clear, the product operand is replaced by Ø 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
```

Operation:

```
tmp1 <- degree;
if tmp1 GTRU 31 then RESERVED OPERAND FAULT;
tmp2 <- tbladdr;
tmp3 <- {(tmp2)+};          !tmp3 accumulates the partial result
                            !tmp3 is of type x
if POLYH then -(SP) <- arg;
while tmp1 GTRU 0 do
        begin               !computation loop
        tmp4 <- {arg * tmp3};   !tmp4 accumulates new partial result.
                                !tmp3 has old partial result.
                !Perform multiply, and retain the 31 (POLYF),
                !63 (POLYD, POLYG), or 127 (POLYH) most significant
                !bits of the fraction by truncating the unnormalized
                !product. (The most significant bit of the 31, 63,
                !or 127 bits in the product magnitude will be zero
                !if the product magnitude is LSS 1/2 and GEQ 1/4.)
                !Use the result in the following add operation.
        tmp4 <- tmp4 + (tmp2);
                !normalize, and round to type x.
                !Check for over/underflow only after the combined
                !multiply/add/normalize/round sequence.
        if OVERFLOW then FLOATING OVERFLOW FAULT
        if UNDERFLOW then
                begin
                if FU EQL 1 then FLOATING UNDERFLOW FAULT;
                tmp4 <- 0;      !force result to 0;
                end;
        tmp1 <- tmp1 - 1;
        tmp2 <- tmp2 + {size of data type};
        tmp3 <- tmp4;   !update partial result in tmp3
        end;
  if POLYF then
        begin
        R0 <- tmp3;
        R1 <- 0;
        R2 <- 0;
        R3 <- tmp2;
        end;
  if POLYD or POLYG then
        begin
        R1'R0 <- tmp3;
        R2 <- 0;
        R3 <- tmp2;
        R4 <- 0;
```

```
                R5 <- Ø;
                end;
        if POLYH then
                begin
                SP <- SP + 16;
                R3'R2'R1'RØ <- tmp3;
                R4 <- Ø;
                R5 <- tmp2;
                end;
```

Condition Codes:

```
        N <- RØ LSS Ø;
        Z <- RØ EQL Ø;
        V <- {floating overflow};
        C <- Ø;
```

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 RØ (R1'RØ for POLYD and POLYG, R3'R2'R1'RØ for POLYH)) are replaced by the result. The result computed is:

```
        if d = degree
        and x = arg
        result = C[Ø] + 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
RØ = result
R1 = Ø
R2 = Ø
R3 = table address +  degree*4 + 4

POLYD and POLYG
RØ = high order part of result
R1 = low order part of result
R2 = Ø
R3 = table address + degree*8 + 8
R4 = Ø
R5 = Ø

POLYH
RØ = 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 = Ø
R5 = table address + degree*16 + 16
```

2. On a floating fault:

   1. If PSL<FPD> = Ø, the instruction faults and all relevant side effects are restored to their original state.

   2. If PSL<FPD> = 1, the instruction is suspended and state is saved in the general registers as follows:

```
POLYF
RØ = tmp3  !partial result after iteration prior to the
                       !one causing the overflow/underflow
R1 = arg
R2<7:Ø> = tmp1      !number of iterations remaining
R2<31:8> = implementation specific
R3 = tmp2  !points to table entry causing exception

POLYD and POLYG
R1'RØ = tmp3        !partial result after iteration prior to
the
              !one causing the overflow/underflow
R2<7:Ø> = 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'RØ = tmp3 !partial result after iteration prior to
                   !the one causing the overflow/underflow
R4<7:Ø> = 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 a fault handler.

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

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

5. On a reserved operand fault:

   1. if PSL<FPD> = Ø, the reserved operand is either the degree operand (greater than 31), or the argument operand, or some coefficient.

   2. if PSL<FPD> = 1, the reserved operand is a coefficient, and R3 (except for POLYH) or R5 (for POLYH) is pointing at the value which caused the exception.

   3. 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 is continuable.

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 non zero 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) = CØ + C1*x + C2*x**2
where CØ = 1.Ø,  C1 = .5,  and C2 = .25
```

```
        POLYF   X,#2,PTABLE
        .
        .
        .
PTABLE: .FLOAT  Ø.25     ;C2
        .FLOAT  Ø.5      ;C1
        .FLOAT  1.Ø      ;CØ
```

SUB Subtract

Format:

```
opcode sub.rx, dif.mx              2 operand

opcode sub.rx, min.rx, dif.wx      3 operand
```

Operation:

```
dif <- dif - sub;        !2 operand

dif <- min - sub;        !3 operand
```

Condition Codes:

```
N <- dif LSS Ø;
Z <- dif EQL Ø;
V <- {floating overflow};
C <- Ø;
```

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 Ø 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
```

Operation:

```
        src - Ø;
```

Condition Codes:

```
        N <- src LSS Ø;
        Z <- src EQL Ø;
        V <- Ø;
        C <- Ø;
```

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. TSTx src is equivalent to CMPx src, #Ø, 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.

4.10 CHARACTER STRING INSTRUCTIONS

A character string is specified by 2 operands:

1. An unsigned word operand which 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 RØ through R1, RØ through R3, or RØ through R5 to contain a control block which 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 and if any is found, the control block is updated, a first part done bit is set in the PSL, and the instruction interrupted (See Chapter 6). After the interruption, the instruction resumes transparently. The format of the control block is:

```
+-----------------------------+-----------------------------+
|                             |          LENGTH 1           | : RØ
+-----------------------------+-----------------------------+
|                        ADDRESS 1                          | : R1
+-----------------------------+-----------------------------+
|                             |          LENGTH 2           | : R2
+-----------------------------+-----------------------------+
|                        ADDRESS 2                          | : R3
+-----------------------------+-----------------------------+
|                             |          LENGTH 3           | : R4
+-----------------------------+-----------------------------+
|                        ADDRESS 3                          | : R5
+-----------------------------------------------------------+
```

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 will not occur when a zero length string is specified because no memory reference occurs.

The following instructions are described in this section.

CMPC   Compare Characters

Format:

```
        opcode len.rw, srcladdr.ab, src2addr.ab  3 operand

        opcode srcllen.rw, srcladdr.ab, fill.rb,
               src2len.rw, src2addr.ab              5 operand
```

Operation:

```
        tmpl <- len;                                !3 operand
        tmp2 <- srcladdr;

        tmp3 <- src2addr;
        if tmpl EQL Ø then; !Condition Codes affected on tmpl EQL Ø
        if tmpl GTRU Ø then
                begin

        while {tmpl NEQU Ø} do
        if (tmp2) EQL (tmp3) then
                        !Condition Codes affected on ((tmp2) EQL (tmp3))

                begin
                tmpl <- tmpl - l;
                tmp2 <- tmp2 + l;
                tmp3 <- tmp3 + l;
                end;

                else exit while loop;

                end;
        RØ <- tmpl;
        Rl <- tmp2;
        R2 <- RØ;
        R3 <- tmp3;

        tmpl <- srcllen;                            !5 operand
        tmp2 <- srcladdr;
        tmp3 <- src2len;
        tmp4 <- src2addr;

        if {tmpl EQL Ø} AND {tmp3 EQL Ø} then;

                !Condition codes affected on {tmpl EQL Ø} AND {tmp3 EQL
 Ø}
        while {tmpl NEQU Ø} AND {tmp3 NEQU Ø} do
        if (tmp2) EQL (tmp4) then

                        !Condition Codes affected on ((tmp2) EQL (tmp4))
                begin
                tmpl <- tmpl - l;
                tmp2 <- tmp2 + l;
                tmp3 <- tmp3 - l;
                tmp4 <- tmp4 + 1;
                end;

        else exit while loop;
        if NOT{tmp1 NEQU Ø} AND {tmp3 NEQU Ø} then
                begin

        while {tmp1 NEQU Ø} AND {(tmp2) EQL fill} do
                        !Condition Codes affected on ((tmp2) EQL fill)
                begin
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 + 1;
                end;

        while {tmp3 NEQU Ø} AND {fill EQL (tmp4)} do

                        !Condition Codes affected on (fill EQL (tmp4))
                begin
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 + 1;
                end;

                end;

        RØ <- tmp1;
        R1 <- tmp2;
        R2 <- tmp3;
        R3 <- tmp4;
```

Condition Codes:

```
        !Final Condition Codes reflect last affecting
        !of Condition Codes in Operation above.
        N <- {first byte} LSS {second byte};
        Z <- {first byte} EQL {second byte};
        V <- Ø;
        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 string 1 specified by the length and address 1 operands are compared with the bytes of string 2 specified by the length and address 2 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 string 1 specified by the length 1 and address 1 operands are compared with the bytes of the string 2 specified by the length 2 and address 2 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 (i.e. Z is set and N, V, and C are cleared).

Notes:

1. After execution of CMPC3:

   RØ = number of bytes remaining in string 1 (including byte which terminated comparison); RØ is zero only if strings are equal

   R1 = address of the byte in string 1 which terminated comparison; if strings are equal, address of one byte beyond string 1

   R2 = RØ

   R3 = address of the byte in string 2 which terminated comparison; if strings are equal, address of one byte beyond string 2.

2. After execution of CMPC5:

   RØ = number of bytes remaining in string 1 (including byte which terminated comparison); RØ is zero only if string 1 and string 2 are of equal length and equal or string 1 was exhausted before comparison terminated

   R1 = address of the byte in string 1 which terminated comparison; if comparison did not terminate before string 1 exhausted, address of one byte beyond string 1

   R2 = number of bytes remaining in string 2 (including byte which terminated comparison); R2 is zero only if string 2 and string 1 are of equal length or string 2 was exhausted before comparison terminated

   R3 = address of the byte in string 2 which terminated comparison; if comparison did not terminate before string 2 was exhausted, address of one byte beyond string 2.

$

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
```

Operation:

```
        tmpl <- len;
        tmp2 <- addr;
        if tmpl GTRU Ø then
                begin
        while {tmpl NEQ Ø} AND {(tmp2) NEQ char}  do
                begin
                tmpl <- tmpl - 1;
                tmp2 <- tmp2 + 1;
                end;
                end;
        RØ <- tmpl;
        R1 <- tmp2;
```

Condition Codes:

```
        N <- Ø;
        Z <- RØ EQL Ø;
        V <- Ø;
        C <- Ø;
```

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:

   RØ = number of bytes remaining in the string (including located one) if byte located; otherwise Ø

   R1 = address of the byte located if byte located; otherwise address of one 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
```

Operation:

```
        tmpl <- objlen;
        tmp2 <- objaddr;
        tmp3 <- srclen;
        tmp4 <- srcaddr;
        tmp5 <- tmpl;

        while {tmpl NEQU Ø} AND {tmp3 GEQU tmpl} do
                begin
                if (tmp2) EQL (tmp4) then
                        begin
                        tmpl <- tmpl - 1;
                        tmp2 <- tmp2 + 1;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 + 1;
                        end
                else
                        begin
                        tmp2 <- tmp2 - ZEXT (tmp5-tmpl);
                        tmp3 <- {tmp3 - 1} + {tmp5-tmpl};
                        tmp4 <- {tmp4 + 1} - ZEXT (tmp5-tmpl);
                        tmpl <- tmp5;
                        end;
                end;

        if {tmp3 LSSU tmpl} then
                begin
                tmp4 <- tmp4 + tmp3;
                tmp3 <- Ø;
                end;

        RØ <- tmpl;
        R1 <- tmp2;
        R2 <- tmp3;
        R3 <- tmp4;
```

Condition Codes:

```
        N <- Ø;
        Z <- RØ EQL Ø;  !match found
        V <- Ø;
        C <- Ø;
```

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 which 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:

   RØ = if a match occurred Ø; otherwise the number of bytes in the object string.

   R1 = if a match occurred, the address of one byte beyond the object string i.e. objaddr + objlen; otherwise the address of the object string.

   R2 = if a match occurred, the number of bytes remaining in the source string; otherwise Ø.

   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 i.e. 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 RØ-R3 are left just as though the substring were found.

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

MOVC Move Character

Format:

```
        opcode len.rw, srcaddr.ab, dstaddr.ab       3 operand

        opcode srclen.rw, srcaddr.ab, fill.rb,
               dstlen.rw, dstaddr.ab                5 operand
```

Operation:

```
        tmpl <- len;                                !3 operand
        tmp2 <- srcaddr;
        tmp3 <- dstaddr;
        if tmp2 GTRU tmp3 then
                begin
                while tmpl NEQU Ø do
                        begin
                        (tmp3) <- (tmp2);
                        tmpl <- tmpl - 1;
                        tmp2 <- tmp2 + 1;
                        tmp3 <- tmp3 + 1;
                        end;
                R1 <- tmp2;
                R3 <- tmp3;
                end
        else
                begin
                tmp4 <- tmpl;
                tmp2 <- tmp2 + ZEXT(tmpl);
                tmp3 <- tmp3 + ZEXT(tmpl);
                while tmpl NEQU Ø do
                        begin
                        tmpl <- tmpl - 1;
                        tmp2 <- tmp2 - 1;
                        tmp3 <- tmp3 - 1;
                        (tmp3) <- (tmp2);
                        end;
                R1 <- tmp2 + ZEXT(tmp4);
                R3 <- tmp3 + ZEXT(tmp4);
                end;
        RØ <- Ø;
        R2 <- Ø;
        R4 <- Ø;
        R5 <- Ø;
        tmp1 <- srclen;                                   !5 operand
        tmp2 <- srcaddr;
        tmp3 <- dstlen;
        tmp4 <- dstaddr;
        if tmp2 GTRU tmp4 then
                begin
                while {tmp1 NEQU Ø} AND {tmp3 NEQU Ø} do
                        begin
                        (tmp4) <- (tmp2);
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 + 1;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 + 1;
                        end;
                while tmp3 NEQU Ø do
                        begin
                        (tmp4) <- fill;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 + 1;
                        end;
                R1 <- tmp2;
                R3 <- tmp4;
                end
        else
                begin
                tmp5 <- MINU(tmp1, tmp3);
                tmp6 <- tmp3;
                tmp2 <- tmp2 + ZEXT(tmp5);
                tmp4 <- tmp4 + ZEXT(tmp6);
                while tmp3 GTRU tmp1 do
                        begin
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 - 1;
                        (tmp4) <- fill;
                        end;
                while tmp3 NEQU  Ø do
                        begin
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 - 1;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 - 1;
                        (tmp4) <- (tmp2);
                        end;
                R1 <- tmp2 + ZEXT (tmp5);
                R3 <- tmp4 + ZEXT (tmp6);
                end;
        RØ <- tmp1;
        R2 <- Ø;
        R4 <- Ø;
        R5 <- Ø;
```

Condition Codes:

```
        N <- Ø;                 !MOVC3
        Z <- 1;
        V <- Ø;
        C <- Ø;

        N <- srclen LSS dstlen; !MOVC5
        Z <- srclen EQL dstlen;
        V <- Ø;
        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:

   RØ = Ø

   R1 = address of one byte beyond the source string

   R2 = Ø

   R3 = address of one byte beyond the destination string.

   R4 = Ø

   R5 = Ø

2. After execution of MOVC5:

   RØ = number of unmoved bytes remaining in source string. RØ is non-zero only if source string is longer than destination string

   R1 = address of one byte beyond the last byte in source string that was moved

   R2 = Ø

   R3 = address of one byte beyond the destination string .

   R4 = Ø

   R5 = Ø

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

4. MOVC5 with a Ø 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
```

Operation:

```
        tmp1 <- srclen;
        tmp2 <- srcaddr;
        tmp3 <- dstlen;
        tmp4 <- dstaddr;
        if tmp2 GTRU tmp4 then
                begin
                while {tmp1 NEQU 0} AND {tmp3 NEQU 0}
                        begin
                        (tmp4) <- (tbladdr + ZEXT((tmp2)));
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 + 1;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 + 1;
                        end;
                while {tmp3 NEQU 0} do
                        begin
                        (tmp4) <- fill;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 + 1;
                        end;
                R1 <- tmp2;
                R5 <- tmp4;
                end;
        else
                begin
                tmp5 <- MINU(tmp1,tmp3);
                tmp6 <- tmp3;
                tmp2 <- tmp2 + ZEXT(tmp5);
                tmp4 <- tmp4 + ZEXT(tmp6);
                while tmp3 GTRU tmp1 do
                        begin
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 - 1;
                        (tmp4) <- fill;
                        end;
                while tmp3 NEQU 0 do
                        begin
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 - 1;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 - 1;
                        (tmp4) <- (tbladdr + ZEXT((tmp2)));
                        end;
                R1 <- tmp2 + ZEXT(tmp5);
                R5 <- tmp4 + ZEXT(tmp6);
                end;
        RØ <- tmpl;
        R2 <- Ø;
        R3 <- tbladdr;
        R4 <- Ø;
```

Condition Codes:

```
        N <- srclen LSS dstlen;
        Z <- srclen EQL dstlen;
        V <- Ø;
        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 and 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 zeroth entry 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:

After execution:

RØ = number of untranslated bytes remaining in source string;
RØ is non-zero only if source string is longer than destination string

R1 = address of one byte beyond the last byte in source string that was translated

R2 = Ø

R3 = address of the translation table.

R4 = Ø

R5 = address of one byte beyond the destination
string.

MOVTUC Move Translated Until Character

Format:

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

Operation:

```
        tmp1 <- srclen;
        tmp2 <- srcaddr;
        tmp3 <- dstlen;
        tmp4 <- dstaddr;

        if tmp1 GTRU Ø and tmp3 GTRU Ø then
                begin

        while {tmp1 NEQU Ø} AND {tmp3 NEQU Ø} do
        if{(tbladdr + ZEXT(tmp2)) NEQU esc} then

                begin
                (tmp4) <- (tbladdr + ZEXT(tmp2));
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 + 1;
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 + 1;
                end;

        else exit while loop;

                end;

        RØ <- tmp1;
        R1 <- tmp2;
        R2 <- Ø;
        R3 <- tbladdr;
        R4 <- tmp3;
        R5 <- tmp4;
```

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 and 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 index into a 256 byte table whose zeroth entry address 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 RØ through R5 are UNPREDICTABLE. If the source and destination strings overlap and their addresses are not identical, the destination string and registers RØ through R5 are UNPREDICTABLE. If the source and destination string addresses are identical, the translation is performed correctly.

Notes:

After execution:

RØ = number of bytes remaining in source string (including the byte which caused the escape). RØ is zero only if the entire source string was translated and moved without escape

R1 = address of the byte which resulted in destination string exhaustion or escape; or if no exhaustion or escape, address of one byte beyond the source string

R2 = Ø

R3 = address of the table

R4 = number of bytes remaining in the destination string

R5 = address of the byte in the destination string which would have received the translated byte which 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 one byte beyond the destination string.

SCANC Scan Characters

Format:

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

Operation:

```
        tmp1 <- len;
        tmp2 <- addr;
        if tmp1 GTRU 0 then
                begin
        while {tmp1 NEQU 0} AND
                {{(tbladdr + ZEXT((tmp2))) AND mask} EQL 0} do
                begin
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 + 1;
                end;
                end;
        R0 <- tmp1;
        R1 <- tmp2;
        R2 <- 0;
        R3 <- tbladdr;
```

Condition Codes:

```
        N <- 0;
        Z <- R0 EQL 0;
        V <- 0;
        C <- 0;
```

Exceptions:

none

Opcodes:

2A	SCANC	Scan Characters

Description:

The bytes of the string specified by the length and address operands are successively used to index into a 256 byte table whose zeroth entry address is specified by the table address operand. The byte selected from the table is ANDed with the mask operand. The operation continues until the result of the AND is non-zero or all the bytes of the string have been exhausted. If a non-zero AND result is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.

Notes:

1. After execution:

   RØ = number of bytes remaining in the string (including the byte which produced the non-zero AND result)
   RØ is zero only if there was no non-zero AND result.

   R1 = address of the byte which produced non-zero AND result; or, if no non-zero result, address of one byte beyond the string

   R2 = Ø

   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
```

Operation:

```
        tmpl <- len;
        tmp2 <- addr;
        if tmpl GTRU Ø then
                begin
        while {tmpl NEQ Ø} AND {(tmp2) EQL char} do
                begin
                tmpl <- tmpl - 1;
                tmp2 <- tmp2 + 1;
                end;
                end;
        RØ <- tmpl;
        R1 <- tmp2;
```

Condition Codes:

```
        N <- Ø;
        Z <- RØ EQL Ø;
        V <- Ø;
        C <- Ø;
```

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:

RØ = number of bytes remaining in the string (including the unequal

one) if unequal byte located; otherwise Ø

R1 = address of the byte located if byte located; otherwise address of one 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 character.

SPANC Span Characters

Format:

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

Operation:

```
tmpl <- len;
tmp2 <- addr;
if tmpl GTRU Ø then
        begin
while {tmpl NEQU Ø} AND
        {{(tbladdr + ZEXT((tmp2))) AND mask} NEQ Ø} do
        begin
        tmpl <- tmpl - 1;
        tmp2 <- tmp2 + 1;
        end;
        end;
RØ <- tmpl;
R1 <- tmp2;
R2 <- Ø;
R3 <- tbladdr;
```

Condition Codes:

```
N <- Ø;
Z <- RØ EQL Ø;
V <- Ø;
C <- Ø;
```

Exceptions:

none

Opcodes:

2B	SPANC	Span Characters

Description:

The bytes of the string specified by the length and address operands are successively used to index into a 256 byte table whose zeroth entry address is specified by the table address operand. The byte selected from the table is ANDed with the mask operand. The operation continues until the result of the AND is zero or 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:

   RØ = number of bytes remaining in the string (including the byte which produced the zero AND result) RØ is zero only if there was no zero AND result.

   R1 = address of the byte which produced a zero AND result; or, if no non-zero result, address of one byte beyond the string

   R2 = Ø

   R3 = address of the table.

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

4.11 CYCLIC REDUNDANCY CHECK INSTRUCTION

This instruction is designed to implement the calculation and checking of a cyclic redundancy check 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-11 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 is such as to minimize the number of undetected block errors of specific lengths. The choice of a CRC polynomial is not given here; see, for example, the article "Cyclic Codes for Error Detection" by W. Peterson and D. Brown in the Proceedings of the IRE (January, 1961).

The operands to the CRC instruction are a string descriptor, a 16-longword table, and an initial CRC. The string descriptor is a standard VAX-11 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 initial CRC is used to start the polynomial correctly. Typically, it has the value Ø or -1, but would be different if the data stream is represented by a sequence of non-contiguous strings.

The CRC instruction operates by scanning the string, and for each byte of the data stream, including it in the CRC being calculated. The byte is included by XORing it to the right 8 bits of the CRC. Then the CRC is shifted right 1 bit, inserting zero on the left. The right most bit of the CRC (lost by the shift) is used to control the XORing of the CRC polynomial with the resultant CRC. If the bit is set, the polynomial is XORed with the CRC. Then the CRC is again shifted right and the polynomial is conditionally XORed with the result a total of eight times. The actual algorithm used can shift by one, two, or four 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 a multiple of eight bits in length. If it is not, the stream must be right adjusted in the string with leading Ø bits.

CRC   Calculate Cyclic Redundancy Check

Format:

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

Operation:

```
        tmp1 <- strlen;
        tmp2 <- stream;
        tmp3 <- inicrc;
        tmp4 <- tbl;
        while tmp1 NEQU Ø do
                begin
                tmp3<7:Ø><- tmp3<7:Ø> XOR (tmp2)+;
                for  tmp5  <-  1,limit  do              !see  note  5  for
limit,s,i
                        tmp3 <- ZEXT(tmp3<31:s>) XOR
                               (tmp4 + {4*ZEXT(tmp3<s-1:Ø>*i)};
                tmp1 <- tmp1 -1;
                end;
        RØ <- tmp3;
        R1 <- Ø;
        R2 <- Ø;
        R3 <- tmp2;      !address of end of string + 1
```

Condition Codes:

```
        N <- RØ LSS Ø;
        Z <- RØ EQL Ø;
        V <- Ø;
        C <- Ø;
```

Exceptions:

none

Opcodes:

ØB	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 and is normally Ø or -1 unless the CRC is calculated in several steps. The result is left in RØ. If the polynomial is less than order-32, the result must be extracted from the result. The CRC polynomial is expressed by the contents of the 16-longword table. See the notes for the calculation of the table.

Notes:

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

2. If the CRC polynomial is less than order 32, the result must be extracted from the low order bits of RØ.

3. The following algorithm can be used to calculate the CRC table given a polynomial expressed as follows:

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

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

```
         SUBROUTINE LIB$CRC_TABLE (POLY, TABLE)

         INTEGER*4 POLY, TABLE(Ø:15), TMP, X

         DO 19Ø INDEX = Ø, 15

         TMP = INDEX
         DO 15Ø I = 1, 4
         X = TMP .AND. 1
         TMP = ISHFT(TMP,-1)     !logical shift right one bit
         IF (X .EQ. 1) TMP =  TMP .XOR. POLY
15Ø      CONTINUE
         TABLE(INDEX) = TMP

19Ø      CONTINUE
         RETURN
         END
```

4. 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:	12ØØØ1 (octal)
initialize:	Ø
result:	RØ<15:Ø>

   CCITT (used in ADCCP, HDLC, SDLC)

polynomial:	$x^16 + x^12 + x^5 + 1$
poly:	1Ø2Ø1Ø (octal)
initialize:	-1<15:Ø>

result:	one's complement of RØ<15:Ø>

AUTODIN-II

polynomial:	x^32+x^26+x^23+x^22+x^16+x^12 +x^11+x^1Ø+x^8+x^7+x^5+x^4+x^2+x+1
poly:	EDB8832Ø (hex)
initialize:	-1<31:Ø>
result:	one's complement of RØ<31:Ø>

5. This instruction produces an UNPREDICTABLE result unless the table is well formed, such as produced in note 3. Note that for any well formed table, entry [Ø] is always Ø and entry[8] is always the polynomial expressed as in note 3. The operation can be implemented using shifts of one, two, or four bits at a time as follows:

shift (s)	steps per byte (limit)	table index	table index multiplier (i)	use table entries
1	8	tmp3<Ø>	8	[Ø]=Ø,[8]
2	4	tmp3<1:Ø>	4	[Ø]=Ø,[4],[8],[12]
4	2	tmp3<3:Ø>	1	all

6. If the stream has zero length, RØ receives the initial CRC.

4.12 DECIMAL STRING INSTRUCTIONS

Decimal string instructions operate on Packed Decimal strings. Convert instructions are provided between Packed Decimal and Trailing Numeric String (Overpunched and Zoned) and Leading Separate Numeric string formats. Where necessary a specific data type is identified. Where the phrase decimal string is used, it means any of the three data types.

A decimal string is specified by 2 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 2).

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. This byte contains a sign for Left Separate Numeric strings. The address is specified by a byte operand of address access type.

Each of the decimal string instructions uses general registers RØ through R3 or RØ through R5 to contain a control block which maintains updated addresses and state during the execution of the instruction. At completion, the registers containing addresses are available to the software to 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 and if any is found, the control block is updated. First Part Done is set in the PSL, and the instruction interrupted (See chapter 6). After the interruption, the instruction resumes transparently. The format of the control block at completion is:

```
 3
 1                                                              Ø
+---------------------------------------------------------------+
|                              Ø                                | : RØ
+---------------------------------------------------------------+
|                          ADDRESS 1                            | : R1
+---------------------------------------------------------------+
|                              Ø                                | : R2
+---------------------------------------------------------------+
|                          ADDRESS 2                            | : R3
+---------------------------------------------------------------+
|                              Ø                                | : R4
+---------------------------------------------------------------+
|                          ADDRESS 3                            | : R5
+---------------------------------------------------------------+
```

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.

4.12.1 Decimal Overflow

Decimal overflow occurs if the destination string is too short to contain all the digits (excluding leading zeroes) 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 -Ø). 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.

4.12.2 Zero Numbers

A zero result has a positive sign for all operations which complete without decimal overflow, except for CVTPT which does not fixup a -Ø to a +Ø. 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 -Ø is treated as identical to a decimal string with value +Ø. Thus for example +Ø compares equal to -Ø. When condition codes are affected on a -Ø result they are affected as if the result were +Ø: i.e., N is cleared and Z is set.

4.12.3 Reserved Operand Exception

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

4.12.4 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, in general, be UNPREDICTABLE. The destination strings, registers used by the instruction and condition codes will, in general, be UNPREDICTABLE when a reserved operand abort occurs.

4.12.5 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 "Ø" digit in the high nibble of the first byte of the string.

A packed decimal string contains an invalid nibble if:

1. A digit occurs in the sign position.

2. A sign occurs in a digit position.

3. For an even length string, a non-zero nibble occurs in the high order nibble of the lowest addressed byte.

4.12.6 Zero Length Decimal Strings

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

The length of a trailing numeric string can be Ø. 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 will 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 Ø.

The length of a leading separate numeric string can be Ø. In this case one 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 identically Ø.

4.12.7 Instruction Descriptions

The following instructions are described in this section.

		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
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, {RØ-3.wl}[1]

16. Subtract Packed 6 Operand[1]
    SUBP6 sublen.rw, subaddr.ab, minlen.rw, minaddr.ab,
    diflen.rw, difaddr.ab, {RØ-5.wl}

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
```

Operation:

```
        ({sumaddr + ZEXT(sumlen/2)} : sumaddr) <-
                ({sumaddr + ZEXT(sumlen/2)} : sumaddr) +
                ({addaddr + ZEXT(addlen/2)} : addaddr); !4 operand

        ({sumaddr + ZEXT(sumlen/2)} : sumaddr) <-
                ({add2addr + ZEXT(add2len/2)} : add2addr) +
                ({add1addr  +  ZEXT(add1len/2)}  :  add1addr);         !6
operand
```

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 addend 1 string specified by the addend 1 length and addend 1 address operands is added to the addend 2 string specified by the addend 2 length and addend 2 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:

   RØ = Ø

   R1 = address of the byte containing the most significant digit of the addend string

   R2 = Ø

   R3 = address of the byte containing the most significant digit of the sum string

2. After execution of ADDP6:

   RØ = Ø

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

   R2 = Ø

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

   R4 = Ø

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

3. The sum string, RØ through R3 (or RØ through 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
```

Operation:

```
        ({dstaddr + ZEXT(dstlen/2)} : dstaddr) <-
               {({srcaddr + ZEXT(srclen/2)} : srcaddr)
                       + {round <3:Ø>*{1Ø ** {-cnt-1}}}}
                       * {1Ø ** cnt} ;
```

Condition Codes:

```
        N <- {dst string} LSS Ø;
        Z <- {dst string} EQL Ø;
        V <- {decimal overflow};
        C <- Ø;
```

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 1Ø 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:

   RØ = Ø

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

   R2 = Ø

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

2. The destination string, RØ through 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:Ø 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 non-zero, or if bits 3:Ø 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 five. Truncation may be accomplished by using a zero round operand.

CMPP Compare Packed

Format:

```
        opcode len.rw, src1addr.ab, src2addr.ab          3 operand

        opcode src1len.rw, src1addr.ab, src2len.rw,
               src2addr.ab                                4 operand
```

Operation:

```
        ({src1addr + ZEXT(len/2)} : src1addr) -
                ({src2addr + ZEXT(len/2)} : src2addr);  !3 operand

        ({src1addr + ZEXT(src1len/2)} : src1addr) -
                ({src2addr  +  ZEXT(src2len/2)}  :  src2addr);        !4
operand
```

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 string 1.

R2 = Ø

R3 = address of the byte containing the most significant digit of string 2.

2. RØ through 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
```

Operation:

```
        ({dstaddr + ZEXT(dstlen/2)} : dstaddr) <- conversion of src;
```

Condition Codes:

```
        N <- {dst string} LSS Ø;
        Z <- {dst string} EQL Ø;
        V <- {decimal overflow};
        C <- Ø;
```

Exceptions:

reserved operand
decimal overflow

Opcodes:

```
  F9    CVTLP   Convert Long to Packed
```

Description:

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

Notes:

1. After execution:

   RØ = Ø

   R1 = Ø

   R2 = Ø

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

2. The destination string, RØ through 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
```

Operation:

```
dst <- conversion of ({srcaddr + ZEXT(srclen/2)} : srcaddr);
```

Condition Codes:

```
N <- dst LSS Ø;
Z <- dst EQL Ø;
V <- {integer overflow};
C <- Ø;
```

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:

   RØ = Ø

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

   R2 = Ø

   R3 = Ø

2. The destination operand, RØ through 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 1 above. Thus RØ through R3 may be used as the destination operand.

4. If the source string has a value outside the range -2,147,483,648 through 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. Thus, 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
```

Operation:

```
        {dst string} <- conversion of {src string};
```

Condition Codes:

```
        N <- {src string} LSS Ø;
        Z <- {src string} EQL Ø;
        V <- {decimal overflow};
        C <- Ø;
```

Exceptions:

reserved operand
decimal overflow

Opcodes:

```
  Ø8     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:

   RØ = Ø

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

   R2 = Ø

   R3 = address of the sign byte of the destination string

2. The destination string, R0 through 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 may 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
```

Operation:

```
        {dst string} <- conversion of {src string};
```

Condition Codes:

```
        N <- {src string} LSS Ø;
        Z <- {src string} EQL Ø;
        V <- {decimal overflow};
        C <- Ø;
```

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 (even if the source string value is -Ø) of the source string (i.e., the byte containing the sign and the least significant digit) as an unsigned index into a 256 byte table whose zeroth entry address is specified by the table address operand. The byte read out of 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:

   RØ = Ø

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

R2 = Ø

R3 = address of the most significant digit of the destination string

2. The destination string, RØ through R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string or the table, the source string or the table contains an invalid nibble, or 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 if and only if the source is non-zero and contains a minus sign.

4. By appropriate specification of the table, conversion to any form of trailing numeric string may be realized. See Chapter 2 for the preferred form of trailing overpunch, zoned and unsigned data. In addition, the table may 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 non-zero packed decimal source string (not including leading zeroes). Conversion of a source string with zero value never results in overflow. Conversion of a non-zero 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
```

Operation:

```
        {dst string} <- conversion of {src string}
```

Condition Codes:

```
        N <- {dst string} LSS Ø;
        Z <- {dst string} EQL Ø;
        V <- {decimal overflow};
        C <- Ø;
```

Exceptions:

```
        reserved operand
        decimal overflow
```

Opcodes:

```
  Ø9     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:

   1. The length of the source Leading Separate numeric string is outside the range Ø through 31.

   2. The length of the destination packed decimal string is outside the range Ø through 31.

   3. The source string contains an invalid byte. An invalid byte is any character other than an ASCII "Ø" through "9" in a digit byte or an ASCII "+", "<space>", or "-" in the sign byte.

2. After execution:

   RØ = Ø

R1 = address of the sign byte of the source string

R2 = Ø

R3 = address of the byte containing the most significant digit of the destination string.

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

CVTTP Convert Trailing Numeric to Packed

Format:

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

Operation:

```
        {dst string} <- conversion of {src string}
```

Condition Codes:

```
        N <- {dst string}LSS Ø;
        Z <- {dst string} EQL Ø;
        V <- {decimal overflow};
        C <- Ø;
```

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 zeroth entry is specified by the table address operand. The byte read out of the table replaces the highest addressed byte of the destination string (i.e. 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:

    1. The length of the source trailing numeric string is outside the range Ø through 31.

    2. The length of the destination packed decimal string is outside the range Ø through 31.

3. The source string contains an invalid byte. An invalid byte is any value other than ASCII "Ø" through "9" in any high order byte (i.e., any byte except the least significant byte).

4. The translation of the least significant digit produces an invalid packed decimal digit or sign nibble.

2. After execution:

   RØ = Ø

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

   R2 = Ø

   R3 = address of the byte containing the most significant digit of the destination string.

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

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

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

6. By appropriate specification of the table, conversion from any form of trailing numeric string may be realized. See Chapter 2 for the preferred form of trailing overpunch, zoned and unsigned data. In addition, the table may 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
```

Operation:

```
        ({quoaddr + ZEXT(quolen/2)} : quoaddr) <-
                ({divdaddr + ZEXT(divdlen/2)} : divdaddr) /
                ({divraddr + ZEXT(divrlen/2)} : divraddr);
```

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 SP is restored to its original contents and the contents of {(SP)-16}:{(SP)-1} are UNPREDICTABLE.

2. The division is performed such that:

    1. The absolute value of the remainder (which is lost) is less that the absolute value of the divisor.

    2. 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.

3. 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 through 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 0 or a reserved operand abort occurs.

MOVP Move Packed

Format:

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

Operation:

```
        ({dstaddr + ZEXT(len/2)} : dstaddr) <-
                ({srcaddr + ZEXT(len/2)} : srcaddr);
```

Condition Codes:

```
        N <- {dst string} LSS Ø;
        Z <- {dst string} EQL Ø;
        V <- Ø;
        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:

   RØ = Ø

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

   R2 = Ø

   R3 = address of the byte containing the most significant digit of the destination string.

2. The destination string, RØ through 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
```

Operation:

```
        ({prodaddr + ZEXT(prodlen/2)} : prodaddr) <-
                ({muldaddr + ZEXT(muldlen/2)} : muldaddr) *
                ({mulraddr + ZEXT(mulrlen/2)} : mulraddr);
```

Condition Codes:

```
        N <- {prod string} LSS Ø;
        Z <- {prod string} EQL Ø;
        V <- {decimal overflow};
        C <- Ø;
```

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:

   RØ = Ø

   R1 = address of the byte containing the most significant digit of the multiplier string

   R2 = Ø

   R3 = address of the byte containing the most significant digit of the multiplicand string

   R4 = Ø

R5 = address of the byte containing the most significant digit of the product string

2. The product string, RØ through 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:

```
        opcode sublen.rw, subaddr.ab, diflen.rw,
               difaddr.ab                              4 operand

        opcode sublen.rw, subaddr.ab, minlen.rw,
               minaddr.ab, diflen.rw, difaddr.ab       6 operand
```

Operation:

```
        ({difaddr + ZEXT(diflen/2)} : difaddr) <-
                ({difaddr + ZEXT(diflen/2)} : difaddr) -
                ({subaddr + ZEXT(sublen/2)} : subaddr); !4 operand

        ({difaddr + ZEXT(diflen/2)} : difaddr) <-
                ({minaddr + ZEXT(minlen/2)} : minaddr) -
                ({subaddr + ZEXT(sublen/2)} : subaddr); !6 operand
```

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 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:

   RØ = Ø

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

   R2 = Ø

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

2. After execution of SUBP6:

   RØ = Ø

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

   R2 = Ø

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

   R4 = Ø

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

3. The difference string, RØ through R3 (RØ through 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.

4.13 EDIT INSTRUCTION

This instruction is designed to implement the common editing functions which occur in handling fixed format output. It operates by converting a packed decimal string to a character string. This operation is exemplified by a MOVE to a numeric editted (PICTURE) item in COBOL or PL/I, but the instruction can be used for other applications as well. The operation consists of converting an input packed decimal number to an output character string, generating characters for the output. When converting digits, options include leading zero fill, leading zero protection, insertion of floating sign, insertion of floating currency symbol, insertion of special sign representations, and blanking an entire field when it is zero.

The operands to the EDITPC instruction are an input packed decimal string descriptor, a pattern specification, and the starting address of the output string. The packed decimal descriptor is a standard VAX-11 operand pair of the length of the decimal string in digits (up to 31) and the starting address of the string. The pattern specification is the starting address of a pattern operation editing sequence which is interpreted much the way that the normal instructions are. The output string is described by only its starting address because the pattern defines the length unambiguously.

While the EDITPC instruction is operating, it manipulates two character registers and the four condition codes. One character register contains the fill character. This is normally an ASCII blank, but would be changed to asterisk for check protection. The other character register contains the sign character. Initially this contains either an ASCII blank or a minus sign depending upon the sign of the input. This can be changed to allow other sign representations such as plus/minus or plus/blank and can be manipulated in order to output special notations such as CR or DB. The sign register can also be changed to the currency sign in order to implement a floating currency sign. After execution, the condition codes contain the sign of the input (N), the presence of a zero source (Z), an overflow condition (V), and the presence of significant digits (C). Condition code N is determined at the start of the instruction and is not changed thereafter (except for correcting a -Ø input). The other condition codes are computed and updated as the instruction proceeds. When the EDITPC instruction terminates, registers RØ-R5 contain the conventional values after a decimal instruction.

EDITPC Edit Packed to Character String

Format:

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

Operation:

```
        if srclen GTRU 31 then {reserved operand abort};
        PSW<V,C> <- 0;
        PSW<Z> <- 1;
        PSW<N> <- {src has minus sign};
        R0 <- srclen;
        tmpl <- R0;
        R1 <- srcaddr;
        R2 <- ??? ' {if PSW<N> EQL 0 then " " else "-"} ' " ";
                                  !<15:8>=sign, <7:0>=fill
        R3 <- pattern;
        R4 <- ???;
        R5 <- dstaddr;
        exit_flag <- false;

        while NOT exit_flag do
                begin
                {fetch pattern byte};
                {if pattern 0:4 no operand};
                {if pattern 40:47 increment R3 and
                        fetch one byte operand};
                {if pattern 80:AF except 80, 90, A0
                        operand is rightmost nibble};
                {else {reserved operand}};
                {perform pattern operator};
                if NOT exit_flag then {increment R3};
                end;

        if R0 NEQ 0 then {reserved operand};
        R0 <- tmpl;              !length of source string
        R1 <- R1 - {tmpl/2}      !point to start of source string
        R2 <- 0;
        R4 <- 0;
        if PSW<Z> EQL 1 then PSW<N> <- 0;
```

Condition Codes:

```
        N <- {src string} LSS 0;                !N <- 0 if src is -0
        Z <- {src string} EQL 0;
        V <- {decimal overflow};                !non-zero 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 editted version of the source string specified by the source length and source address operands. The editting is performed according to the pattern string starting at the address pattern and extending until a pattern end (EO$END) pattern operator is encountered. The pattern string consists of one byte pattern operators. Some pattern operators take no operands. Some take a repeat count which is contained in the rightmost nibble of the pattern operator itself. The rest take a one byte operand which follows the pattern operator immediately. This operand is either an unsigned integer length or a byte character. The individual pattern operators are described on the following pages.

Notes:

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

2. The destination string is UNPREDICTABLE if the source string contains an invalid nibble, if the EO$ADJUST_INPUT operand is outside the range 1 through 31, if the source and destination strings overlap, or if the pattern and destination strings overlap.

3. After execution:

   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 one byte beyond the last byte of the destination string

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

4. If V is set at the end and DV is enabled, 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 -Ø, the result may be -Ø 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 Ø or is outside the destination string.

8. If more input digits are requested by the pattern than are specified, then a reserved operand abort is taken with RØ = -1 and R3 = location of pattern operator which requested the extra digit. The condition codes and other registers are as specified in note 11. This abort is not continuable.

9. If fewer input digits are requested by the pattern than are specified, then 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 is not continuable.

1Ø. 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 is continuable as long as the defined register state is manipulated according to the pattern operator description and the state specified as ??? is preserved.

11. On a reserved operand exception as specified in notes 8 through 1Ø, FPD is set and the condition codes and registers are as follows:

```
N  = {src has minus sign}

Z  = all source digits Ø so far

V  = non-zero digits lost

C  = significance

RØ = -zeros<15:Ø> ' remaining srclen<15:Ø>

R1 = current source location

R2 = ??? ' sign ' fill

R3 = location of edit pattern operator causing exception
```

R4 = ???

R5 = location of next destination byte

where:

zeros = count of source zeros to supply

sign = current contents of sign character register

fill = current contents of fill character register

Summary of EDIT pattern operators

	name	operand	summary
insert:			
	EO$INSERT	ch	insert character, fill if insignificant
	EO$STORE_SIGN	-	insert sign
	EO$FILL	r	insert fill
move:			
	EO$MOVE	r	move digits, filling insignificant
	EO$FLOAT	r	move digits, floating sign
	EO$END_FLOAT	-	end floating sign
fixup:			
	EO$BLANK_ZERO	len	fill backward when zero
	EO$REPLACE_SIGN	len	replace with fill if -Ø
load:			
	EO$LOAD_FILL	ch	load fill character
	EO$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:			
	EO$SET_SIGNIF	-	set significance flag
	EO$CLEAR_SIGNIF	-	clear significance flag
	EO$ADJUST_INPUT	len	adjust source length
	EO$END	-	end edit

where:

ch	=	one character
r	=	repeat count in the range 1 through 15
len	=	length in the range 1 through 255

EDIT pattern operator encoding

```
 (hex)

  ØØ        EO$END
  Ø1        EO$END_FLOAT
  Ø2        EO$CLEAR_SIGNIF
  Ø3        EO$SET_SIGNIF
  Ø4        EO$STORE_SIGN

 Ø5..1F     Reserved to DEC

 2Ø..3F     Reserved for all time

  4Ø        EO$LOAD_FILL        \
  41        EO$LOAD_SIGN         |
  42        EO$LOAD_PLUS         |-- character is in next byte
  43        EO$LOAD_MINUS        |
  44        EO$INSERT           /

  45        EO$BLANK_ZERO       \
  46        EO$REPLACE_SIGN      |-- unsigned length is in next byte
  47        EO$ADJUST_INPUT     /

 48..5F     Reserved to DEC

 6Ø..7F     Reserved to CSS, customers

8Ø,9Ø,AØ    Reserved to DEC
 81..8F     EO$FILL             \
 91..9F     EO$MOVE              |-- repeat count is <3:Ø>
 A1..AF     EO$FLOAT            /

 BØ..FE     Reserved to DEC
   FF       Reserved for all time
```

The following 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 through 15, an unsigned byte length (len), or a character byte (ch). In the formal descriptions, the following two routines are invoked:

```
READ:                     !function value Ø through 9
        if RØ EQL Ø then {reserved operand};

        if RØ LSS Ø then
                begin
                READ <- Ø;
                RØ<31:16> <- RØ<31:16> + 1;     !see EO$ADJUST_INPUT
                end;
        else
                begin
                READ <- (R1)<3+4*RØ<Ø>:4*RØ<Ø>>; !get next nibble
                                        !alternating high then low
                RØ <- RØ - 1;
                if RØ<Ø> EQL 1 then R1 <- R1 + 1;
                end;
        return;


STORE(char):
        (R5) <- char;
        R5 <- R5 + 1;
        return;
```

Also the following definitions are used:

```
        fill =  R2<7:Ø>

        sign =  R2<15:8>
```

EO$INSERT Insert Character

Purpose:

Insert a fixed character, substituting the fill character if not significant

Format:

```
        pattern     ch
```

Operation:

```
        if PSW<C> EQL 1 then STORE(ch) else STORE(fill);
```

Pattern operators:

44	EO$INSERT	Insert Character

Description:

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

Notes:

This pattern operator is used for blankable inserts (e.g., comma) and fixed inserts (e.g., slash). Fixed inserts require that significance be set (by EO$SET_SIGNIF or EO$END_FLOAT).

EO$STORE_SIGN Store Sign

Purpose:

Insert the sign character

Format:

```
        pattern
```

Operation:

```
        STORE(sign);
```

Pattern operators:

Ø4	EO$STORE_SIGN	Store Sign

Description:

The contents of the sign register is placed into the destination.

Notes:

This pattern operator is used for any non-floating arithmetic sign. It should be preceded by a EO$LOAD_PLUS and/or EO$LOAD_MINUS if the default sign convention is not desired.

EO$FILL Store Fill

Purpose:

Insert the fill character

Format:

```
        pattern     r
```

Operation:

```
        repeat r do STORE(fill);
```

Pattern operators:

8x	EO$FILL	Store Fill

Description:

The right nibble of the pattern operator is the repeat count. The contents of the fill register is placed into the destination repeat times.

Notes:

This pattern operator is used for fill (blank) insertion.

EO$MOVE Move Digits

Purpose:

Move digits, filling for insignificant digits (leading zeros)

Format:

```
        pattern     r
```

Operation:

```
        repeat r do
                begin
                tmp <- READ;
                if tmp NEQU Ø then
                        begin
                        PSW<Z> <- Ø;
                        PSW<C> <- 1;     !set significance
                        end;
                if PSW<C> EQL Ø then STORE(fill)
                        else STORE("Ø" + tmp);
                end;
```

Pattern operators:

9x	EO$MOVE	Move Digits

Description:

The right nibble of the pattern operator is the repeat count. For repeat times, the following algorithm is executed. The next digit is moved from the source to the destination. If the digit is non-zero, significance is set and zero is cleared. If the digit is not significant (i.e., 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. This pattern operator is used 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$FLOAT Float Sign

Purpose:

Move digits, floating the sign across insignificant digits

Format:

```
        pattern      r
```

Operation:

```
        repeat r do
                begin
                tmp <- READ;
                if tmp NEQU Ø then
                        begin
                        if PSW<C> EQL Ø then
                                begin
                                STORE(sign);
                                PSW<Z> <- Ø;
                                PSW<C> <- 1;     !set significance
                                end;
                        end;
                if PSW<C> EQL Ø then STORE(fill)
                        else STORE("Ø" + tmp);
                end;
```

Pattern operators:

Ax	EO$FLOAT	Float Sign

Description:

The right nibble of the pattern operator is the repeat count. For repeat times, the following algorithm is executed. The next digit from the source is examined. If it is non-zero and significance is not yet set, then the contents of the sign register is 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 is 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. This pattern operator is used to move digits with a floating arithmetic sign. The sign must already be setup 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. This pattern operator is used to move digits with a floating currency sign. The sign must already be setup with a 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$END_FLOAT End Floating Sign

Purpose:

End a floating sign operation

Format:

```
        pattern
```

Operation:

```
        if PSW<C> EQL Ø then
                begin
                STORE(sign);
                PSW<C> <- 1;     !set significance
                end;
```

Pattern operators:

Ø1	EO$END_FLOAT	End Floating Sign

Description:

If the floating sign has not yet been placed in the destination (i.e., if significance is not set), the contents of the sign register is stored in the destination and significance is set.

Notes:

This pattern operator is used after a sequence of one or more EO$FLOAT pattern operators which start with significance clear. The EO$FLOAT sequence can include intermixed EO$INSERTs and EO$FILLs.

EO$BLANK_ZERO Blank Backwards When Zero

Purpose:

Fixup the destination to be blank when the value is zero

Format:

```
        pattern     len
```

Operation:

```
if len EQLU 0 then {UNPREDICTABLE};
if PSW<Z> EQL 1 then
        begin
        R5 <- R5 - len;
        repeat len do STORE(fill);
        end;
```

Pattern operators:

```
  45    EO$BLANK_ZERO   Blank Backwards When Zero
```

Description:

The 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 is stored into the last length bytes of the destination string.

Notes:

1. The length must be non-zero 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. This pattern operator is used 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$REPLACE_SIGN Replace Sign When Zero

Purpose:

Fixup the destination sign when the value is zero

Format:

```
        pattern     len
```

Operation:

```
        if len EQLU 0 then {UNPREDICTABLE};
        if PSW<Z> EQL 1 then (R5 - len) <- fill;
```

Pattern operators:

46	EO$REPLACE_SIGN Replace Sign When Zero

Description:

The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero (i.e., if Z is set), then the contents of the fill register is stored into the byte of the destination string which is length bytes before the current position.

Notes:

1. The length must be non-zero 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. This pattern operator can be used 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$LOAD_ Load Register

Purpose:

Change the contents of the fill or sign register

Format:

```
        pattern     ch
```

Operation:

```
                        !select one depending on pattern operator

        fill <- ch;             !EO$LOAD_FILL

        sign <- ch;             !EO$LOAD_SIGN

        if PSW<N> EQL 0 then sign <- ch;        !EO$LOAD_PLUS

        if PSW<N> EQL 1 then sign <- ch;        !EO$LOAD_MINUS
```

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. EO$LOAD_FILL is used to setup check protection (* instead of space).

2. EO$LOAD_SIGN is used to setup a floating currency sign.

3. EO$LOAD_PLUS is used to setup a non-blank plus sign.

4. EO$LOAD_MINUS is used to setup a non-minus minus sign (such as CR, DB, or the PL/I +).

EO$_SIGNIF Significance

Purpose:

Control the significance (leading zero) indicator

Format:

pattern

Operation:

```
        PSW<C> <- 0;              !EO$CLEAR_SIGNIF

        PSW<C> <- 1;              !EO$SET_SIGNIF
```

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

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

EO$ADJUST_INPUT Adjust Input Length

Purpose:

Handle source strings with lengths different from the output

Format:

```
        pattern     len
```

Operation:

```
        if len EQLU Ø or len GTRU 31 then {UNPREDICTABLE};
        if RØ<15:Ø> GTRU len
        then
                begin
                RØ<31:16> <- Ø
                repeat RØ<15:Ø> - len do
                        if READ NEQU Ø then
                                begin
                                PSW<Z> <- Ø;
                                PSW<C> <- 1;    !set significance
                                PSW<V> <- 1;
                                end;
                end;
        else RØ<31:16> <- RØ<15:Ø> - len;       !negative of number to
fill
```

Pattern operators:

47	EO$ADJUST_INPUT Adjust Input Length

Description:

The pattern operator is followed by an unsigned byte integer length in the range 1 through 31. If the source string has more digits than this length, the excess leading digits are read and discarded. If any discarded digits are non-zero then 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 RØ<31:16>.

Notes:

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

EO$END End Edit

Purpose:

End the edit operation

Format:

```
        pattern
```

Operation:

```
        exit_flag <- true;                  !terminate edit loop
                                            !end processing is
instruction                                 !described    under    EDITPC
```

Pattern operators:

```
  00     EO$END           End Edit
```

Description:

The edit operation is terminated.

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.

4.14 OTHER VAX-11 INSTRUCTIONS

The following instructions are specified in other chapters of this document as indicated below.

	Instructions
1. Chapter 5:	
Probe {Read, Write} Accessability PROBE{R,W} mode.rb, len.rw, base.ab	2
2. Chapter 6:	
Change Mode CHM{K,E,S,U} param.rw, {-(ySP).w*} Where y=MINU(x, PSL<current_mode>)	4
Return from Exception or Interrupt REI {(SP)+.r*}	1
3. Chapter 7:	
Load Process Context LDPCTX {PCB.r*, -(KSP).w*}	1
Save Process Context SVPCTX {(SP)+.r*, PCB.w*}	1
4. Chapter 9:	
Move To Process Register MTPR src.rl, procreg.rl	1
Move From Processor Register MFPR procreg.rl, dst.wl	1

BUG Bugcheck

Format:

```
        opcode message.bx
```

Operation:

```
        {fault to report error}
```

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 faults. The VAX/VMS operating system treats these as requests to report software detected errors. The in-line message identifier is zero extended to a longword (BUGW) and interpreted as a condition value (see Appendix C, VAX/VMS Run Time Library Reference Manual). If the process is privileged to report bugs, a log entry is made. If the process is not privileged, a reserved instruction is signalled.

CHAPTER 5
MEMORY MANAGEMENT

17-Jun-81 -- Rev 5.3

5.1 INTRODUCTION

Memory management consists of the hardware and software which control the allocation and use of physical memory. Typically, in a multiprogramming system, several processes may reside in physical memory at the same time. The VAX-11 uses memory protection and multiple address spaces to ensure that one process will not affect other processes or the operating system.

To further improve software reliability, four hierarchical access modes provide memory access control. They are, from most to least privileged: kernel, executive, supervisor, and user. Protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write for each of the four access modes. Any location accessible to one mode is also accessible to all more privileged modes. Furthermore, for each access mode, any location that can be written can also be read.

The CPU generates virtual addresses when an image is executed. However, before these addresses can be used to access instructions and data, they must be translated into physical addresses. Memory management software maintains tables of mapping information (page tables) that keep track of where each 512-byte virtual page is located in physical memory. The CPU utilizes this mapping information when it translates virtual addresses to physical addresses.

Therefore, memory management is the scheme that provides both the memory protection and memory mapping mechanisms of the VAX-11. The memory management meets several development goals:

1. Provide a large address space for instructions and data.

2. Allow data structures up to one gigabyte.

3. Provide convenient and efficient sharing of instructions and data.

4. Contribute to software reliability.

A virtual memory system provides a large address space, yet allows programs to run on hardware with small memory configurations. Programs execute in an environment termed a process. The virtual memory system for VAX-11 provides each process with a 4 billion byte address space.

The virtual address space is divided into two equal size spaces, the system address space and the per-process address space. The system address space is the same for all processes. It contains the operating system which is written as callable procedures. Thus all system code can be available to all other system and user code via a simple CALL. Each process has its own separate process address space. However, several processes may have access to the same page, thus providing controlled sharing.

5.2 VIRTUAL ADDRESS SPACE

A virtual address is a 32 bit unsigned integer specifying a byte location in the address space. The programmer sees a linear array of 4,294,967,296 bytes. The virtual address space is broken into 512 byte units termed pages. The page is the unit of relocation and protection.

This virtual address space is too large to be contained in any presently available main memory. Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. Memory management also provides page protection between processes. The operating system controls the virtual-to-physical address mapping tables, and swaps the inactive but used parts of the virtual address space onto the external storage media.

The virtual address space is divided into two parts. The half with the smaller addresses, known as "per-process space," is distinct for each process running on the system. The half with the larger addresses, known as "system space," is shared by all processes. Virtual address space is illustrated in Figure 5-1.

```
           +-----------------------------------------+
00000000   |                                         |
           |              length of PØ Region        |
           |              in pages (PØLR)            |
           |                                         |     p
           |   PØ                                    |     e
           | (Program)    ---------------------      |     r
           |  Region                                 |     -
           |                  | PØ Region            |     P
           |                  | growth direction     |     r
           |                  |                      |     o
           |3FFFFFFF          v                      |     c
           +-----------------------------------------+     e
           |40000000          ^                      |     s
           |                  |                      |     s
           |                  | Pl Region            |
           |                  | growth direction     |     S
           |   Pl             |                      |     p
           | (Control)    ---------------------      |     a
           |  Region                                 |     c
           |              length of Pl Region        |     e
           |              in pages (2**21-PlLR)      |
7FFFFFFF   |                                         |
      -----+-----------------------------------------+------
80000000   |                                         |
           |                                         |
           |              length of System Region    |
           |  S           in pages (SLR)             |
           |  Region      ---------------------      |
           |                                         |     S
           |                  |                      |     y
           |                  | System Region        |     s
           |                  | growth direction     |     t
           |BFFFFFFF          v                      |     e
           +-----------------------------------------+     m
           |C0000000                                 |
           |                                         |     S
           |                                         |     p
           |                                         |     a
           |              Reserved                   |     c
           |              Region                     |     e
           |                                         |
           |                                         |
FFFFFFFF   |                                         |
           +-----------------------------------------+
```

Figure 5-1
Virtual Address Space

5.2.1 Process Space

The smaller-addressed half (addresses 00000000-7FFFFFFF, hex) of the virtual address space is termed "per-process space." The per-process space is divided into two equal parts, the program region (P0 region) and the control region (P1 region). Each process has a separate address translation map for per-process space, so the per-process spaces of all processes are completely disjoint (see the section on Sharing at the end of this chapter). The address map for per-process space is context switched (changed) when the process running on the system is changed (see the chapter on Process Structure).

5.2.2 System Space

The larger-addressed half (addresses 80000000-FFFFFFFF, hex) of the virtual address space is termed "system space." All processes use the same address translation map for system space, so system space is shared among all processes. The address map for system space is not context switched.

5.2.3 Virtual Address Format

The VAX-11 processor generates a 32-bit virtual address for each instruction and operand in memory. As the process executes, the system translates each virtual address to a physical address. The virtual address has the following format:

```
 3
 1                                              9 8                0
+-----------------------------------------------+------------------+
|                       VPN                     |     byte  #      |
+-----------------------------------------------+------------------+
```

Figure 5-2
Virtual Address Format

VPN	<31:9>	The Virtual Page Number field specifies the virtual page to be referenced. The virtual address space contains 8,388,608 (2**23) pages of 512 bytes each.
Byte #	<8:0>	The byte number field specifies the byte address within the page. A page contains 512 bytes.

When bit 31 is one, the address is in the system space. When bit 31 is zero, the address is in the per-process space.

Within the per-process space, bit 30 distinguishes between the program and control regions. When bit 30 is one, the control region is referenced, and when it is zero, the program region is referenced.

5.2.4 Virtual Address Space Layout

The layout of virtual address space is illustrated in Figure 5-1. Note that access to each of the three regions (PØ, P1, System) is controlled by a length register (PØLR, P1LR, SLR). Within the limits set by the length registers, the access is further controlled by page tables that specify the validity, access requirements, and physical location of each page in the memory.

5.3 MEMORY MANAGEMENT CONTROL

The action of translating a virtual address to a physical address is governed by the setting of the Memory Mapping Enable (MME) bit in the MAPEN internal processor register. Figure 5-3 illustrates the privileged MAP ENable register.

```
 3
 1
+                                                             1 Ø
+-------------------------------------------------------------+-+
|                                                             |M|
|                            MBZ                              |M|
|                                                             |E|
+-------------------------------------------------------------+-+
```

Figure 5-3
MAP ENable Register (MAPEN)
(to read: MFPR #56, dst.wl)
(to write: MTPR src.rl, #56)

MAPEN<Ø> is the Memory Mapping Enable (MME) bit. When MME is set to 1, memory management is enabled. When MME is set to Ø, memory management is disabled. At processor initialization time, MAPEN is initialized to Ø.

5.3.1 Memory Management Disabled

Setting MME to Ø turns off address translation and access control. Virtual address bit n, VA<n>, is copied directly to the corresponding physical address bit, PA<n>, for n = Ø to 29. VA<31:3Ø> are ignored; PA<31:3Ø> are always zero. VA<n> is ignored if PA<n> doesn't exist. (The number of PA bits is implementation dependent.)

PA = VA<29:Ø> modulo (2** number of PA bits)

There is no page protection: all accesses are allowed in all modes. No modify bit is maintained.

5.4 ADDRESS TRANSLATION

When MME is a 1, address translation and access control are on. The processor uses the following to determine whether an intended access is allowed:

1. The virtual address, which is used to index a page table,

2. The intended access type (read or write), and

3. The current privilege level from the Processor Status Longword, or Kernel level for page table mapping references.

If the access is allowed and the address can be mapped, the result is the physical address corresponding to the specified virtual address.

The intended access is READ if the operation to be performed is a read. The intended access is WRITE if the operation to be performed is a write. If the operation to be performed is a modify (that is, read followed by write) the intended access is specified as a WRITE.

If an operand is an address operand, then no reference is made. Hence the page need not be accessible and need not even exist.

5.4.1 Page Table Entry (PTE)

The CPU uses a Page Table Entry (PTE) to translate virtual addresses to physical addresses. Figure 5-4a illustrates the PTE format.

```
 3 3        2 2 2 2 2 2 2 2
 1 Ø        7 6 5 4 3 2 1 Ø                                    Ø
+-+--------+-+-+---+-+-+---------------------------------------+
|V| PROT   |M|Z|OWN|S|S|                  PFN                  |
+-+--------+-+-+---+-+-+---------------------------------------+
```

Figure 5-4a
Page Table Entry

V	<31>	Valid bit - governs the validity of the M bit and PFN field. V=1 for valid; V=Ø for not valid. When V=Ø, the M and PFN fields are reserved for DIGITAL software.
PROT	<3Ø:27>	PROTection field - this field is always valid and is used by the CPU hardware even when V=Ø.
M	<26>	Modify bit - When the Valid bit is clear, M is not used by CPU hardware, and is reserved for DIGITAL software and I/O devices. When the Valid bit is set, M shows whether the page has been modified. If M is clear, the page has not

been modified. If M is set, the page may have been modified.

M is cleared only by software. It is set by CPU hardware on a successful write or modify to the page. In addition, it may be set by the probe-write instruction (PROBEW) or by an implied probe-write. M is not set if the page is inaccessible. Beyond that, it is UNPREDICTABLE whether M is set if a fault occurs in an instruction which would otherwise have modified the page.

For example, if a write reference crosses a page boundary where the first page is not accessible and the second page is accessible, the reference will fault. M is unchanged in the PTE mapping the first page. It is UNPREDICTABLE whether M is set in the PTE mapping the second page.

It is UNPREDICTABLE whether the modification of a process PTE<M> bit causes modification of the system PTE that maps that process page table. Note that the update of the M bit is not interlocked in a multiprocessor system.

OWN	<24:23>	OWNer bits - reserved for DIGITAL software use as the access mode of the owner of the page (that is, the mode allowed to alter the page protection or to delete the page); not examined or altered by any hardware.
PFN	<20:0>	Page Frame Number - the upper 21 bits of the physical address of the base of the page. Used by CPU hardware only if V=1.
Z	<25>	Zero bit - bit 25,is RESERVED to DIGITAL and must be zero. The hardware does not necessarily test that this bit is zero because the PTE is established by privileged software.
S	<22:21>	Software bits - bits 22, and 21 are reserved for DIGITAL software.

(Software symbols defined for the above fields use PTE$ as the prefix.)

The operating system software uses some combinations of the software bits to implement its page management data structures and functions. Among the functions implemented this way are initialize-pages-with-zeros, copy-on-reference, page sharing, and transitions between active and swapped-out states. VAX/VMS encodes these functions in PTEs whose Valid bit, PTE<31>, is a 0 and processes them whenever a page fault occurs.

5.4.2 Page Table Entry (PTE) For I/O Devices

Some I/O devices, such as the DR32, use VAX-11 memory management to translate addresses. These I/O devices use a Page Table Entry format which is an extension of that in Figure 5-4a used by the CPU. The extended PTE implements for I/O hardware some functions that the CPU does with software using software bits and page faults. In particular, PTE bits 31, 26, and 22 are decoded into four combinations. Some of these are used in the same way as in the CPU PTE format, and some are used in different ways. The four combinations are:

PTE<31,26,22>			PTE Type
1	x	x	Valid PFN
Ø	Ø	Ø	Valid PFN
Ø	Ø	1	Global Page Table Index
Ø	1	x	Invalid, I/O abort

and their interpretations are:

PTE<31,26,22>=1xx, Figure 5-4b. PTE<2Ø:Ø> is a valid PFN field. This is identical to the PFN field illustrated in Figure 5-4a for the CPU PTE.

```
 3 3       2 2 2 2 2 2 2 2
 1 Ø       7 6 5 4 3 2 1 Ø                                            Ø
+-+-------+-+-+---+-+-+-------------------------------------------+
|1| PROT  |M|Z|OWN|S|S|                    PFN                      |
+-+-------+-+-+---+-+-+-------------------------------------------+
```

Figure 5-4b
PTE<31,26,22>=1xx, Valid PFN

PTE<31,26,22>=ØØØ, Figure 5-4c. PTE<2Ø:Ø> is a valid PFN field. This is identical to the PFN field illustrated in Figure 5-4a for the CPU PTE.

```
 3 3       2 2 2 2 2 2 2 2
 1 Ø       7 6 5 4 3 2 1 Ø                                            Ø
+-+-------+-+-+---+-+-+-------------------------------------------+
|Ø| PROT  |Ø|Z|OWN|Ø|S|                    PFN                      |
+-+-------+-+-+---+-+-+-------------------------------------------+
```

Figure 5-4c
PTE<31,26,22>=ØØØ, Valid PFN

PTE<31,26,22>=ØØ1, Figure 5-4d. PTE<21:Ø> is a Global Page Table Index (GPTX). The I/O device has a Global page table Base Register (GBR) which is loaded by software with a system virtual address. The I/O device calculates GBR + GPTX * 4 to get the system virtual address of a second PTE. The second PTE must contain a valid PFN, and must have PTE<31,26,22> equal to either ØØØ or 1xx, binary. If either of these requirements is not met, the result is UNDEFINED. For those devices that use it, the PROTection field always comes from the first PTE.

```
 3 3     2 2 2 2 2 2 2
 1 Ø     7 6 5 4 3 2 1                                               Ø
+-+-------+-+-+---+-+-----------------------------------------------+
|Ø|  PROT |Ø|Z|OWN|1|                   GPTX                        |
+-+-------+-+-+---+-+-----------------------------------------------+
```

Figure 5-4d
PTE<31,26,22>=ØØ1, Global Page Table Index

PTE<31,26,22>=Ø1x, Figure 5-4e. This PTE format is RESERVED to DIGITAL. I/O devices will abort in a DEVICE DEPENDENT manner.

```
 3 3     2 2 2 2 2 2 2 2
 1 Ø     7 6 5 4 3 2 1 Ø                                             Ø
+-+-------+-+-+---+-+-+---------------------------------------------+
|Ø| PROT  |1|Z|OWN|S|S|                  S                          |
+-+-------+-+-+---+-+-+---------------------------------------------+
```

Figure 5-4e
PTE<31,26,22>=Ø1x, Invalid, I/O abort

I/O devices may look at and check the PROTection field or modify the M bit; this is DEVICE DEPENDENT. Those devices that do use them, use them the same way the CPU does.

I/O devices that do memory mapping use the same SPT as the CPU, but they have their own copies of the SBR and SLR. Buffer addresses are described in terms of a system virtual address of the PTE for the first buffer page and a byte offset within that page. In addition the I/O devices use a Global Page Table in memory and an I/O hardware Global page table Base Register (GBR) which must be loaded by software.

5.4.3 Changes To Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, VMS sets and clears the valid bit and changes the PFN field as pages are swapped in and out.

The software must guarantee that each PTE is always consistent within itself. Changing a PTE one field at a time may give incorrect system operation. An example would be to set PTE<V> with one instruction before establishing PTE<PFN> with another. An interrupt routine between the two instructions could use an address that would map using the inconsistent PTE. The software can solve this problem by building a new PTE in a register and then moving the new PTE to the page table with a single instruction such as MOVL.

Multiprocessing makes the problem more complicated. Another processor, be it another CPU or an I/O processor, can reference the same page tables that the first CPU is changing. The second processor must always read consistent PTEs. In order to guarantee this, first note that PTEs are longwords, longword-aligned. Then two requirements must be met:

1. Whenever the software modifies a PTE in more than one byte, it must use a longword, longword-aligned, write-destination instruction, such as MOVL, and

2. The hardware must guarantee that a longword, longword-aligned write is an "atomic" operation. That is, a second processor cannot read (or write over) any of the first processor's partial results.

5.5 ACCESS CONTROL

Access control is the function of validating whether a particular type of memory access is to be allowed to a particular page. Access to each page is controlled by a protection code that specifies for each access mode whether or not read or write references are allowed. Additionally, each address is checked to make certain that it lies within the PØ, P1, or system region.

5.5.1 Processor Modes

In the order of most privileged to least privileged, the four processor modes are:

Ø - Kernel - used by the kernel of the operating system for page management, scheduling, and I/O drivers.

1 - Executive - used for many of the operating system service calls, including the record management system.

2 - Supervisor - used for such services as command interpretation.

3 - User - used for user level code, utilities, compilers, debuggers, etc.

The access mode of a running process is the current processor mode, stored in the Current Mode field of the Processor Status Longword (PSL) (see the Chapter on Exceptions and Interrupts).

5.5.2 Protection Code

Every page in the virtual address space is protected according to its use. Even though all of the system space is shared, in that the program may generate any address, the program may be prevented from modifying, or even accessing portions of it. A program may also be prevented from accessing or modifying portions of per-process space.

For example, in system space, scheduling queues are highly protected, whereas library routines may be executable by code of any privilege. Similarly per-process accounting information may be in per-process space, but highly protected, while normal user code in per-process spaces is executable at low privilege.

Associated with each page is a protection code that describes the accessibility of the page for each processor mode. The code allows a choice of protection for each processor mode, within the following limits:

1. Each level's access can be read-write, read-only, or no-access.

2. If any level has read access then all more privileged levels also have read access.

3. If any level has write access then all more privileged levels also have write access.

The protection codes for the 15 combinations of page protection are encoded in a 4 bit field in the Page Table Entry as follows:

CODE		MNEMONIC	PRIVILEGE LEVEL				COMMENT
DECIMAL	BINARY		K	E	S	U	
0	0000	NA	-	-	-	-	no ACCESS
1	0001			UNPREDICTABLE			RESERVED
2	0010	KW	RW	-	-	-	
3	0011	KR	R	-	-	-	
4	0100	UW	RW	RW	RW	RW	ALL ACCESS
5	0101	EW	RW	RW	-	-	
6	0110	ERKW	RW	R	-	-	
7	0111	ER	R	R	-	-	
8	1000	SW	RW	RW	RW	-	
9	1001	SREW	RW	RW	R	-	
10	1010	SRKW	RW	R	R	-	
11	1011	SR	R	R	R	-	
12	1100	URSW	RW	RW	RW	R	
13	1101	UREW	RW	RW	R	R	
14	1110	URKW	RW	R	R	R	
15	1111	UR	R	R	R	R	

Key

-	no access	K	Kernel
R	read only	E	Executive
RW	read write	S	Supervisor
		U	User

Figure 5-5
Protection Mnemonics

(Software symbols are defined by using PTE$K_ as a prefix to the above mnemonics.)

This encoding was chosen to simplify hardware access checking for implementations not using a table decoder. The access is allowed if:

```
{CODE NEQU 0} AND
    {{CODE EQLU 4} OR {CM LSSU WM} OR {READ AND {CM LEQU RM}}}

    CM is current processor mode
    RM is left 2 bits of code
    WM is one's complement of right 2 bits of code
```

5.5.3 Length Violation

Every valid virtual address lies within bounds determined by the addressing region (PØ, Pl, or System) and the associated length register (PØLR, PlLR, or SLR). Virtual addresses outside these bounds cause a length violation. The addressing bounds algorithm is a simple limit check whose formal notation is:

```
case VAddr<31:3Ø>
    set
    [Ø]:                                             !PØ region
        if ZEXT( VAddr<29:9> ) GEQU PØLR
            then {length violation};
    [1]:                                             !P1 region
        if ZEXT( VAddr<29:9> ) LSSU P1LR
            then {length violation};
    [2]:                                             !S region
        if ZEXT( VAddr<29:9> ) GEQU SLR
            then {length violation};
    [3]:                                             !reserved region
        {length violation};
    tes;
```

5.5.4 Access Control Violation Fault

An access control fault occurs if an illegal access is attempted, as determined by the current PSL mode and the page's protection field, or if the address causes a length violation.

5.5.5 Access Across A Page Boundary

If an access is made across a page boundary, the order in which the pages are accessed is UNPREDICTABLE. However, for a given page, access control violation always takes precedence over translation not valid.

5.5.6 System Space Address Translation

A virtual address with <31:3Ø>=2 is an address in the system virtual address space.

```
 3 3 2
 1 Ø 9                                            9 8               Ø
+---+---------------------------------------------+-----------------+
| 2 |                                             |      Byte #     |
|<-------System Virtual Page No. (SVPN)------>|                 |
+---+---------------------------------------------+-----------------+
```

Figure 5-6
System Virtual Page Format

The system virtual address space is defined by the System Page Table (SPT), which is a vector of Page Table Entries (PTEs). The SPT is always located in physical address space. The base address of the SPT is also a physical address and is contained in the System Base Register (SBR). The size of the SPT in longwords (that is, the number of PTEs) is contained in the System Length Register (SLR). The SBR points to the first PTE in the SPT. In turn, this PTE maps the first page of System Space, that is, virtual byte address 8ØØØØØØØ(hex).

The PTEs in the System Page Table contain the mapping information themselves, or point to the mapping information in the Global Page Table if the PTE is in GPTX format. (See the section on PTEs for I/O devices for a description of the GPTX format.)

```
 3 3 2
 1 Ø 9                                                        2 1 Ø
+---+---------------------------------------------------------+---+
|MBZ|                 Physical Longword Address               |MBZ|
+---+---------------------------------------------------------+---+
```

Figure 5-7
System Base Register (SBR)
(to read: MFPR   #12, dst.wl)
(to write: MTPR   src.rl, #12)

```
 3                   2 2
 1                   2 1                                       Ø
+--------------------+------------------------------------------+
|        MBZ         |        Length of SPT in longwords        |
+--------------------+------------------------------------------+
```

Figure 5-8
System Length Register (SLR)
(to read: MFPR   #13, dst.wl)
(to write: MTPR   src.rl, #13)

Bits <31:9> of the virtual address contain the Virtual Page Number. However, system virtual addresses have VAddr<31:3Ø>=2. Thus, there could be as many as 2**21 pages in the system region. (Typically the value is in the range of a few hundred to a few thousand system pages; see the section at the end of this chapter on Sharing.) The length field in the System Length Register requires 22 bits to express the values Ø through 2**21 inclusive. At processor initialization time, the contents of both registers are UNPREDICTABLE. Figure 5-9 illustrates the translation of a system virtual address to a physical address.

```
                         3 3 2
                         1 Ø 9                       9 8        Ø
                        +---+-------------------------+---------+
SVA:                    | 2 |                         |  byte   |
(System Virtual         +---+-------------------------+---------+
   Address)                 |    Extract and          |         |
                  3        2|2   Check Length         |         |
                  1        3|2                       2|1Ø       |
                  +---------+-------------------------+--+      |
                  |   Ø     |                         | Ø|      |
                  +---------+-------------------------+--+      |
                                                                |
                                  Add                           |
                                                                |
                                                                |
                  +-----------------------------------+--+      |
SBR:              |   Phys Base Adr of SPT            | Ø|      |
                  +-----------------------------------+--+      |
                                                                |
                                 Yields                         |
                                                                |
                  +-----------------------------------+--+      |
                  |        Phys Adr of PTE            | Ø|      |
                  +-----------------------------------+--+      |
                                                                |
                                  Fetch                         |
               3 3       2 2                                    |
               1 Ø       1 Ø                        Ø           |
              +-+---------+-------------------------+           |
PTE:          |1|         |           PFN           |           |
              +-+---------+-------------------------+           |
             check access |                         |           |
                          |                         |           |
                     3 3|2                          |           |
                     1 Ø|9                         9|8          V Ø
                    +---+---------------------------+-----------+
Physical Adr of Data: | Ø |                         |           |
                    +---+---------------------------+-----------+
```

Figure 5-9
System Virtual to Physical Translation

The algorithm to generate a physical address from a system region virtual address is:

```
    SYS_PA = (SBR+4*SVA<29:9>)<2Ø:Ø>'SVA<8:Ø>      !System Region
```

Note

For all occurrences within this chapter, the parentheses indicate "contents of," the angle brackets indicate referenced bits, and the apostrophe indicates concatenation.

5.5.7 Process Space Address Translation

The process virtual address space is divided into two equal sized, separately mapped regions. If virtual address bit 30 is 0, the address is in region P0. If virtual address bit 30 is a 1, the address is in region P1.

The P0 region maps a virtually contiguous area that begins at the smallest address (0) in the process virtual space and grows in the direction of larger addresses.

P0 is typically used for program images and can grow dynamically.

The P1 region maps a virtually contiguous area that begins at the largest address (2**31 - 1) in the process virtual space and grows in the direction of smaller addresses.

P1 is typically used for system-maintained, per-process context. It may grow dynamically for the user stack.

Each region is described by a virtually contiguous vector of Page Table Entries. Unlike the System Page Table, which is addressed with a physical address, these two page tables are addressed with virtual addresses in the system region of the virtual address space. Thus, for per-Process Space, the address of the PTE is a virtual address in System Space and the fetch of the PTE is simply a longword fetch using a system virtual address.

There is a significant reason to address process page tables in virtual rather than physical space. A physically addressed process page table that required more than a page of PTEs (that is, that mapped more than 64K bytes of process virtual space) would require physically contiguous pages. Such a requirement would make dynamic allocation of process page table space very awkward since a running system tends to fragment storage into page-sized areas.

A process space address translation that causes a translation buffer miss will cause one memory reference for the process PTE. If the virtual address of the page containing the process PTE is also missing from the translation buffer, a second memory reference is required.

When a process Page Table Entry is fetched, a reference is made to System Space. This reference is made as a kernel read. Thus the system page containing a process page table is either "No Access" (that is, protection code zero) or will be accessible (protection code non-zero). Similarly, a check is made against the System page table Length Register (SLR). Thus, the fetch of an entry from a process page table can result in access or length violation faults (see the section on Faults and Parameters).

5.5.8 PØ Region

The PØ region of the address space is mapped by the PØ Page Table (PØPT) which is defined by the PØ Base Register (PØBR) and the PØ Length Register (PØLR). The PØBR contains a virtual address in the system region which is the base address of the PØ Page Table. Figure 5-1Ø illustrates the PØ Base Register. The PØLR contains the size of the PØPT in longwords, that is, the number of Page Table Entries. Figure 5-11 illustrates the PØ Length Register. The Page Table Entry addressed by the PØ Base Register maps the first page of the PØ region of the virtual address space, that is, virtual byte address Ø.

The PTEs in the PØ Page Table contain the mapping information themselves, or point to the mapping information in the Global Page Table if the PTE is in GPTX format. (See the section on PTEs for I/O devices for a description of the GPTX format.)

```
 3 3 2
 1 Ø 9                                                      2 1 Ø
+---+-------------------------------------------------------+---+
| 2 |            System Virtual Longword Address            |MBZ|
+---+-------------------------------------------------------+---+
```

Figure 5-1Ø
PØ Base Register (PØBR)
(to read: MFPR   #8, dst.wl)
(to write: MTPR   src.rl, #8)

```
 3        2 2   2 2 2 2
 1        7 6   4 3 2 1                                      Ø
+---------+-----+---+-----------------------------------------+
|   MBZ   | IGN |MBZ|        Length of PØPT in longwords      |
+---------+-----+---+-----------------------------------------+
```

Figure 5-11
PØ Length Register (PØLR)
(to read: MFPR   #9, dst.wl)
(to write: MTPR   src.rl, #9)

The Virtual Page Number is contained in bits <29:9> of the virtual address. A 22-bit length field is required to express the values Ø through 2**21 inclusive. There could be as many as 2**21 pages in the PØ region.

PØLR<26:24> are ignored on MTPR and read back Ø on MFPR. At processor initialization time, the contents of both registers are UNPREDICTABLE. An attempt to load PØBR with a value less than 2**31 or greater than 2**31 + 2**3Ø - 4 results in a reserved operand fault in some implementations. Figure 5-12 illustrates the PØ virtual address to physical address translation.

```
                          3 3 2
                          1 Ø 9                      9 8       Ø
                         +---+----------------------+---------+
PVA:                     | Ø |                      |  byte   |
(Process Virtual         +---+----------------------+---------+
   Address)                  |                      |         |
                     3      2|2   Extract and       |         |
                     1      3|2   Check Length     2|1Ø       |
                     +--------+----------------------+--+     |
                     |   Ø    |                      | Ø|     |
                     +--------+----------------------+--+     |
                                                              |
                                    Add                       |
                                                              |
                     +---------------------------------+--+   |
PØBR:                |  Sys Virt Base Adr of PØPT      | Ø|   |
                     +---------------------------------+--+   |
                                                              |
                                  Yields                      |
                                                              |
                     +---------------------------------+--+   |
                     |     Sys Virt Adr of PTE         | Ø|   |
                     +---------------------------------+--+   |
                                                              |
                        Fetch by System Space                 |
                          translation algorithm,              |
                          including length and                |
                          Kernel mode access checks           |
                                                              |
                   3 3       2 2                              |
                   1 Ø       1 Ø                    Ø         |
                  +-+--------+----------------------+         |
PTE:              |1|        |         PFN          |         |
                  +-+--------+----------------------+         |
               check access  | this access check    |         |
                             | in current mode      |         |
                             |                      |         |
                        3 3|2                       |         |
                        1 Ø|9                      9|8        V Ø
                       +---+-----------------------+---------+
Physical Adr of Data:  | Ø |                       |         |
                       +---+-----------------------+---------+
```

Figure 5-12
PØ Virtual to Physical Translation

The algorithm to generate a physical address from a PØ region virtual address is:

```
PVA_PTE  = PØBR+4*PVA<29:9>                           !PØ Region
PTE_PA   = (SBR+4*PVA_PTE<29:9>)<2Ø:Ø>'PVA_PTE<8:Ø>
PROC_PA  = (PTE_PA)<2Ø:Ø>'PVA<8:Ø>
```

5.5.9 P1 Region

The P1 region of the address space is mapped by the P1 Page Table (P1PT) which is defined by the P1 Base Register (P1BR) and the P1 Length Register (P1LR). Because P1 space grows towards smaller addresses, and because a consistent hardware interpretation of the base and length registers is desirable, P1BR and P1LR describe the portion of P1 space that is NOT accessible. Figure 5-13 illustrates the P1 Base Register. Figure 5-14 illustrates the P1 Length Register. Note that P1LR contains the number of nonexistent PTEs. P1BR contains a virtual address of what would be the PTE for the first page of P1, that is, virtual byte address 4ØØØØØØØ(hex).

The address in P1BR is not necessarily an address in System Space, but all the addresses of PTEs must be in System Space.

The PTEs in the P1 Page Table contain the mapping information, or point to the mapping information in the Global Page Table if the PTE is in GPTX format. (See the section on PTEs for I/O devices for a description of the GPTX format.)

```
 3
 1                                                       2 1 Ø
+-------------------------------------------------------+---+
|                Virtual Longword Address               |MBZ|
+-------------------------------------------------------+---+
```

Figure 5-13
P1 Base Register (P1BR)
(to read: MFPR #1Ø, dst.wl)
(to write: MTPR src.rl, #1Ø)

```
 3 3              2 2
 1 Ø              2 1                                      Ø
+-+---------------+-----------------------------------------+
|I|      MBZ      |    2**21 - Length of P1PT in longwords  |
+-+---------------+-----------------------------------------+
```

Figure 5-14
P1 Length Register (P1LR)
(to read: MFPR #11, dst.wl)
(to write: MTPR src.rl, #11)

P1LR<31> is ignored on MTPR and reads back Ø on MFPR. At processor initialization time, the contents of both registers are UNPREDICTABLE. An attempt to load P1BR with a value less than 2**31 - 2**23 (7F8ØØØØØ, hex) or greater than 2**31 + 2**3Ø - 2**23 - 4 results in a reserved operand fault in some implementations.

```
                        3 3 2
                        1 0 9                     9 8       0
                       +---+---------------------+---------+
PVA:                   | 1 |                     |  byte   |
(Process Virtual       +---+---------------------+---------+
   Address)                |   Extract and       |         |
                  3       2|2  Check Length      |         |
                  1       3|2                   2|10       |
                  +--------+---------------------+--+       |
                  |    0   |                     | 0|       |
                  +--------+---------------------+--+       |
                                                            |
                                   Add                      |
                                                            |
                  +------------------------------+--+       |
P1BR:             |    Sys Virt Adr of P1PT      | 0|       |
                  +------------------------------+--+       |
                                                            |
                                 Yields                     |
                                                            |
                  +------------------------------+--+       |
                  |    Sys Virt Adr of PTE       | 0|       |
                  +------------------------------+--+       |
                                                            |
                     Fetch by System Space                  |
                       translation algorithm,               |
                       including length and                 |
                       Kernel mode access checks            |
                                                            |
                3 3       2 2                               |
                1 0       1 0                  0            |
               +-+--------+--------------------+            |
PTE:           |1|        |         PFN        |            |
               +-+--------+--------------------+            |
         check access     | this access check  |            |
                          | in current mode    |            |
                          |                    |            |
                     3 3|2                     |            |
                     1 0|9                    9|8         V 0
                    +---+---------------------+---------+
Physical Adr of Data: | 0 |                     |         |
                    +---+---------------------+---------+
```

Figure 5-15
P1 Virtual to Physical Translation

The algorithm to generate a physical address from a P1 region virtual address is:

```
     PVA_PTE  = P1BR+4*PVA<29:9>                            !P1 Region
     PTE_PA   = (SBR+4*PVA_PTE<29:9>)<20:0>'PVA_PTE<8:0>
     PROC_PA  = (PTE_PA)<20:0>'PVA<8:0>
```

5.6 TRANSLATION BUFFER

In order to save actual memory references when repeatedly referencing the same pages, a hardware implementation may include a mechanism to remember successful virtual address translations and page states. Such a mechanism is termed a translation buffer.

When the process context is loaded with LDPCTX, the translation buffer is automatically updated (that is, the process virtual address translations are invalidated). However, when the software changes any part of a valid Page Table Entry for the system or a current process region, it must also move a virtual address within the corresponding page to the Translation Buffer Invalidate Single (TBIS) register with the MTPR instruction. Figure 5-16 illustrates the TBIS register.

Additionally, when the software changes a System Page Table Entry which maps any part of the current process page table, all process pages so mapped must be invalidated in the translation buffer. They may be invalidated by moving an address within each such page into the TBIS register. They may also be invalidated by clearing the entire translation buffer. This is done by moving Ø to the Translation Buffer Invalidate All (TBIA) register with the MTPR instruction. Figure 5-17 illustrates the TBIA register.

The translation buffer must not store invalid PTEs. Therefore, the software is not required to invalidate translation buffer entries when making changes for PTEs that are already invalid.

When the location or size of the system map is changed (SBR, SLR) the entire translation buffer must be cleared.

Whenever Memory Management Enable (MME) is a Ø, the contents of the translation buffer are UNPREDICTABLE. Therefore, before enabling memory management at processor initialization time, or any other time, the entire translation buffer must be cleared.

```
 3                                                             Ø
 1
+---------------------------------------------------------------+
|                       Virtual Address                         |
+---------------------------------------------------------------+
```

Figure 5-16
Translation Buffer Invalidate Single (TBIS)
(to write: MTPR src.rl, #58)

```
 3
 1                                                            0
+-------------------------------------------------------------+
|                            MBZ                              |
+-------------------------------------------------------------+
```

Figure 5-17
Translation Buffer Invalidate All (TBIA)
(to write: MTPR   src.rl, #57)

An internal processor register is available for interrogating the presence of a valid translation in the translation buffer. When a virtual address is written to the TBCHK register with a MTPR instruction, the condition code V bit is set if the translation buffer holds a valid translation for that virtual page. The specification of the TBCHK register is based on VAX/VMS usage. The TBCHK register is reserved for Digital use. Its specification is subject to change without prior notice.

5.7 FAULTS AND PARAMETERS

Two types of faults are associated with memory mapping and protection (see the chapter on Exceptions and Interrupts for a description of faults). A Translation Not Valid Fault is taken when a read or write reference is attempted through an invalid PTE (PTE<31>=0). An Access Control Violation Fault is taken when the protection field of the PTE indicates that the intended page reference in the specified access mode would be illegal. Note that these two faults have distinct vectors in the System Control Block. If both faults could occur, then the Access Control Violation Fault takes precedence. An Access Control Violation Fault is also taken if the virtual address referenced is beyond the end of the associated page table. Such a "length violation" is essentially the same as referencing a PTE that specifies "No Access" in its protection field. To avoid having the fault software recompute the length check, a "length violation" indication is stored in the fault parameter word illustrated in Figure 5-18.

```
 3
 1                                                       2 1 0
+-----------------------------------------------------+-+-+-+
|                          0                          |M|P|L|  :(SP)
+-----------------------------------------------------+-+-+-+
|          some virtual address in the faulting page        |
+-----------------------------------------------------------+
|                 PC of faulting instruction                |
+-----------------------------------------------------------+
|                   PSL at time of fault                    |
+-----------------------------------------------------------+
```

Figure 5-18
Fault Parameter Block

The same parameters are stored for both types of fault. The first parameter pushed on the stack after the PSL and PC is some virtual address in the same page with the virtual address that caused the fault. A Process Space reference can result in a System Space virtual reference for the PTE. If the PTE reference faults, the virtual address that is saved is the process virtual address. In addition, a 1 is stored in bit 1 of the fault parameter word if the fault occurred in the per-process PTE reference.

The second parameter pushed on the Kernel stack contains the following information:

L	<Ø>	Length Violation. Set to 1 to indicate that an Access Control Violation was the result of a length violation rather than a protection violation. This bit is always Ø for a Translation Not Valid Fault.
P	<1>	PTE Reference - Set to 1 to indicate that the fault occurred during the reference to the process page table associated with the virtual address. This can be set on either length or protection faults.
M	<2>	Write or Modify Intent - Set to 1 to indicate that the program's intended access was a write or modify. This bit is Ø if the program's intended access was a read.

5.8 PRIVILEGED SERVICES AND ARGUMENT VALIDATION

5.8.1 Changing Access Modes

Four instructions allow a program to change its access mode to a more privileged mode and transfer control to a service dispatcher for the new mode.

CHMK	change mode to Kernel
CHME	change mode to Exec
CHMS	change mode to Super
CHMU	change mode to User

These instructions, described in detail in the chapter on Exceptions and Interrupts, provide the normal mechanism for less privileged code to call more privileged code. When the mode transition takes place, the previous mode is saved in the Previous Mode field of the PSL, thus allowing the more privileged code to determine the privilege of its caller.

5.8.2 Validating Address Arguments (PROBE instructions)

Two instructions, PROBER and PROBEW, allow privileged services to check addresses passed as parameters. To avoid protection holes in the system, a service routine must always verify that its less privileged caller could have directly referenced the addresses passed as parameters (see the appendix on Address Validation Rules). The PROBE instructions do this verification.

PROBEx PROBE ACCESSIBILITY

Purpose:
verify that arguments can be accessed

Format:
```
opcode  mode.rb, len.rw, base.ab
```

Operation:

```
probe_mode <- MAXU (mode<1:Ø>, PSL<PRV_MOD>)
condition codes <- {accessibility of base} and
                   {accessibility of {base+ZEXT(len)-1}}
                   using probe_mode
```

Condition Codes:

```
N <- Ø;
Z <- if {both accessible} then Ø else 1;
V <- Ø;
C <- C;
```

Exceptions:

translation not valid

Opcodes:

ØC	PROBER	Probe Read Accessibility
ØD	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 between the two end bytes if they will be accessed.

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

Example:

```
MOVL    4(AP),RØ        ;Copy the address of first arg so that
                        ; it can't be changed.
PROBER  #Ø,#4,(RØ)      ;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),RØ        ;Copy length and address of buffer args
                                ; so that they can't change.
        PROBEW  #Ø,RØ,(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.
```

Flows:

The following flows describe the operation of PROBE on each of the virtual addresses it is checking. Note that probing an address returns only the accessibility of the page(s) 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.

1. Look up the virtual address in the translation buffer. If found, use the associated protection field to determine the accessibility and EXIT.

2. Check for length violation for System or per-Process address as appropriate. See elsewhere in this chapter for the length violation check flows. If length violation then return No Access and EXIT.

3. If System virtual address, form physical address of PTE, fetch the PTE, use the protection field to determine the accessibility and EXIT.

4. For per-Process virtual address, must do a virtual memory reference for the PTE.

    1. Look up the virtual address of the PTE in the translation buffer, form the physical address of the PTE if found, fetch the PTE, use the protection field to determine the accessibility and EXIT.

    2. Check the System virtual address of the PTE for length violation. If length violation, then return No Access and EXIT.

    3. T1 <- Page Table Entry for the page containing the per-process PTE.

    4. If the protection field of T1 indicates no access (not even readable by Kernel), then return No Access and EXIT. A no access, not valid pointer to a page of PTE's conserves

storage space for a page full of no access, not valid PTE's.

5. If the valid bit in T1 is Ø, then take a Translation Not Valid Fault and EXIT. This case allows for the demand paging of per-process page tables.

6. Finally, calculate the physical address of the per-process PTE from the PFN field of T1 (see the section on System Space Address Translation), fetch the PTE, use the protection field to determine the accessibility, and EXIT.

5.8.3 Notes On The PROBE instructions

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 PROBEW. If the Valid bit is clear, the Modify bit is not changed.

2. Except for 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 Ø then 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, then 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.

CHAPTER 6
EXCEPTIONS AND INTERRUPTS

12-Dec-8Ø -- Rev 7.1

6.1 INTRODUCTION

At certain times during the operation of a system, events within the system require the execution of particular pieces of software outside the explicit flow of control. The processor transfers control by forcing a change in the flow of control from that explicitly indicated in the currently executing process.

Some of the events are relevant primarily to the currently executing process, and normally invoke software in the context of the current process. The notification of such events is termed an exception.

Other events are primarily relevant to other processes, or to the system as a whole, and are therefore serviced in a system-wide context. The notification process for these events is termed an interrupt, and the system-wide context is described as "executing on the interrupt stack" (IS). Further, some interrupts are of such urgency that they require high-priority service, while others must be synchronized with independent events. To meet these needs, the processor has priority logic that grants interrupt service to the highest priority event at any point in time. The priority associated with an interrupt is termed its interrupt priority level (IPL).

6.1.1 Processor Interrupt Priority Levels (IPL)

The processor has 31 interrupt priority levels (IPL), divided into 15 software levels (numbered, in hex, Ø1 to ØF), and 16 hardware levels (1Ø to 1F, hex). User applications, system calls, and system services all run at process level, which may be thought of as IPL Ø. Higher numbered interrupt levels have higher priority, that is to say, any requests at an interrupt level higher than the processor's current IPL will interrupt immediately but requests at a lower or equal level are deferred.

Interrupt levels Ø1 through ØF (hex) exist entirely for use by software. No device can request interrupts on those levels, but software can force an interrupt by executing MTPR src,#SIRR. (See Chapter 9 and section on software generated interrupts later in this chapter). Once a software interrupt request is made, it will be cleared by the hardware when the interrupt is taken.

Interrupt levels 1Ø to 17 (hex) are for use by devices and controllers, including UNIBUS devices; UNIBUS levels BR4 to BR7 correspond to VAX-11 interrupt levels 14 to 17 (hex).

Interrupt levels 18 to 1F (hex) are for use by urgent conditions, including the interval clock, serious errors, and power fail.

6.1.2 Interrupts

The processor arbitrates interrupt requests according to priority. Only when the priority of an interrupt request is higher than the current IPL (Bits 2Ø:16 of the Processor Status Longword) will the processor raise the IPL and service the interrupt request. The interrupt service routine is entered at the IPL of the interrupt request and will not usually change the IPL set by the processor. Note that this is different from the PDP-11 where the interrupt vector specifies the IPL for the ISR.

Interrupt requests can come from devices, controllers, other processors, or the processor itself. Software executing in kernel mode can raise and lower the priority of the processor by executing MTPR src, #IPL where src contains the new priority desired; see Chapter 9. However, a processor cannot disable interrupts on other processors. Furthermore the priority level of one processor does not affect the priority level of the other processors. Thus in multiprocessor systems interrupt priority levels cannot be used to synchronize access to shared resources. Even the various urgent interrupts including those exceptions that run at IPL 1F (hex) do so on only one processor, thus special software action is required to stop other processors in a multiprocessor system.

6.1.3 Exceptions

Most exception service routines execute at IPL 0 in response to exception conditions caused by the software. A variation from this is serious system failures, which raise IPL to the highest level (1F, hex) to minimize processor interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions, however nested exceptions can occur.

A trap is an exception condition that occurs at the end of the instruction that caused the exception. Therefore the PC saved on the stack is the address of the next instruction that would normally have been executed. Any software can enable and disable some of the trap conditions with a single instruction; see the BISPSW and BICPSW instructions described in Chapter 4.

A fault is an exception condition that occurs during an instruction, and that leaves the registers and memory in a consistent state such that elimination of the fault condition and restarting the instruction will give correct results. Note that faults do not always leave everything as it was prior to the faulted instruction, they only restore enough to allow restarting. Thus, the state of a process that faults may not be the same as that of a process that was interrupted at the same point.

An abort is an exception condition that occurs during an instruction, and potentially leaves the registers and memory indeterminate, such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone.

6.1.4 Contrast Between Exceptions And Interrupts

Generally exceptions and interrupts are very similar. When either is initiated, both the processor status (PSL) and the program counter (PC) are pushed onto the stack. However there are seven important differences:

1. An exception condition is caused by the execution of the current instruction while an interrupt is caused by some activity in the computing system that may be independent of the current instruction.

2. An exception condition is usually serviced in the context of the process that produced the exception condition, while an interrupt is serviced independently from the currently running process.

3. The IPL of the processor is usually not changed when the processor initiates an exception, while the IPL is always raised when an interrupt is initiated.

4. Exception service routines usually execute on a per-process stack while interrupt service routines normally execute on a per-CPU stack.

5. Enabled exceptions are always initiated immediately no matter what the processor IPL is, while interrupts are held off until the processor IPL drops below the IPL of the requesting interrupt.

6. Most exceptions can not be disabled. However, if an exception causing event occurs while that exception is disabled, no exception is initiated for that event even when enabled subsequently. This includes overflow which is the only exception whose occurrence is indicated by a condition code (V). If an interrupt condition occurs while it is disabled, or the processor is at the same or higher IPL, the condition will eventually initiate an interrupt when the proper enabling conditions are met if the condition is still present.

7. The previous mode field in the PSL is always set to Kernel on an interrupt, but on an exception it indicates the mode of the exception.

6.2 PROCESSOR STATUS

When an exception or an interrupt is serviced, the processor status must be preserved so that the interrupted process may continue normally. Basically, this is done by automatically saving the Program Counter (PC) and the Processor Status Longword (PSL). These are later restored with the Return from Exception or Interrupt instruction (REI). Any other status required to correctly resume an interruptable instruction is stored in the general registers. Process context such as the mapping information is not saved or restored on each interrupt or exception. Instead, it is saved and restored only when process context switching is performed. Refer to the LDPCTX and SVPCTX instructions in chapter 7. Other processor status is changed even less frequently; refer to the privileged register descriptions in chapter 9.

The Processor Status Longword (PSL) is a longword consisting of a word of privileged processor status concatenated with the Processor Status Word (PSW). Refer to chapter 2 for a description of the PSW. The PSL is automatically saved on the stack when an exception or interrupt occurs and is saved in the PCB on a process context switch. The PSL can also be stored by the MOVPSL instruction; refer to chapter 4. (The terms current PSL and saved PSL are used to distinguish between this status information when it is in the processor and when copies of it are materialized in memory.)

Bits <31:21> of the current PSL can be changed explicitly only by executing a return from exception or interrupt instruction (REI). REI considers the current mode when restoring the PSL, and faults if a program attempts to increase its privilege by this means. Thus REI is available to all software including user exception handling routines.

```
 3 3 2 2 2 2 2 2 2 2 2 2       1 1
 1 Ø 9 8 7 6 5 4 3 2 1 Ø       6 5               8 7 6 5 4 3 2 1 Ø
+-+-+---+-+-+---+---+-+---------+---------------+-+-+-+-+-+-+-+-+
|C|T|   |F|I|CUR|PRV|M|         |               |D|F|I|T|N|Z|V|C|
|M|P|MBZ|P|S|MOD|MOD|B|   IPL   |      MBZ      |V|U|V| | | | | |
| | |   |D| |   |   |Z|         |               | | | | | | | | |
+-+-+---+-+-+---+---+-+---------+---------------+-+-+-+-+-+-+-+-+
                                 \                               /
                                  \                             /
                                   +-----------PSW-------------+
```

Processor Status Longword

At bootstrap time, PSL is cleared except for IPL and IS.

Bits	Description
3:0	Condition Codes: N, Z, V, C (See chapter 2)
4	Trace enable (T). When set at the beginning of an instruction, causes TP to be set. When TP is set at the end of an instruction, a trace fault is taken before the execution of the next instruction. When TP is clear, no trace exception occurs. Most programs should treat T as UNPREDICTABLE because it is set by debuggers and trace programs for tracing and for proceeding from a breakpoint.
5	Integer Overflow trap enable (IV). When set, forces an integer overflow trap after execution of an instruction that produced an integer result that overflowed or had a conversion error. When IV is clear, no integer overflow trap occurs. (However, the condition code V bit is still set.)
6	Floating Underflow exception enable (FU). When set, forces a floating underflow exception after execution of the instruction that produced an underflowed result (i.e., a result exponent, after normalization and rounding, less than the smallest representable exponent for the data type). When FU is clear, no exception occurs. On the original VAX-11/780 a trap occurs; on all other VAX processors a fault occurs.
7	Decimal Overflow trap enable (DV). When set, forces a decimal overflow trap after execution of an instruction that produced an overflowed decimal (numeric string, or packed decimal) result (i.e., no room to store a non-zero digit) or had a conversion error. When DV is clear, no trap occurs. (However, the condition code V bit is still set.)
15:8	Reserved to DIGITAL, must be zero.
20:16	Interrupt Priority Level (IPL). The current processor priority, in the range 0 to 1F (hex). The processor will accept interrupts only on levels greater than the current level. At bootstrap time, IPL is initialized to 1F (hex).
21	Reserved to DIGITAL, must be zero.
22:23	Previous Access Mode (PRV_MOD). Loaded from current mode by exceptions and CHMx instructions, cleared by interrupts, and restored by REI.
25:24	Current Access Mode (CUR_MOD). The access mode of the currently executing process, as follows: 0 - KERNEL 1 - EXECUTIVE 2 - SUPERVISOR 3 - USER

26 Interrupt Stack (IS). When set the processor is executing on the interrupt stack. Any mechanism that sets IS also clears current mode and raises IPL above 0. If an REI attempts to restore a PSL with IS=1 and non-zero current mode or zero IPL, a reserved operand fault is taken. When clear, the processor is executing on the stack specified by current mode. At bootstrap time, IS is set.

27 First Part Done (FPD). When set, execution of the instruction addressed by PC cannot simply be started at the beginning, and must be restarted at some other, implementation specific, point in its operation. If FPD is set and the exception or interrupt service routine modifies FPD, the general registers, or the saved PSL (except for T or TP), the results of the restarted instruction's execution are UNPREDICTABLE. If a routine sets FPD, the results are also UNPREDICTABLE. However, if software is simulating unimplemented instructions, it may make free use of FPD in its simulation. If the hardware encounters a reserved instruction with FPD set, a reserved instruction fault is taken with the saved PSL<FPD> set.

29:28 Reserved to DIGITAL, must be zero.

30 Trace Pending (TP). Forces a trace fault when set at the beginning of any instruction. Set by the processor if T is set at the beginning of an instruction. Any exception or interrupt service routine clearing TP must also clear T or the tracing of the interrupted instruction, if any, is UNPREDICTABLE.

31 Compatibility Mode (CM). When set the processor is in PDP-11 compatibility mode (see chapter 10). When CM is clear, the processor is in native mode.

6.3 INTERRUPTS

The processor services interrupt requests between instructions. The processor also services interrupt requests at well defined points during the execution of long, iterative instructions such as the string instructions. For these instructions, in order to avoid saving additional instruction state in memory, interrupts are initiated when the instruction state can be completely contained in the registers, PSL, and PC.

The following events cause interrupts:

1. Device completion (IPL 10-17 hex)

2. Device error (IPL 10-17 hex)

3. Device alert (IPL 10-17 hex)

4. Device memory error (IPL 10-17 hex)

5. Console terminal transmit and receive (IPL 14 hex)

6. Interval timer (IPL 18 hex)

7. Recovered memory or bus or processor errors (implementation specific, IPL 18 to 1D hex); The VAX-11/780 processor interrupts at 1B on memory errors.

8. Unrecovered memory or bus or processor errors (implementation specific, IPL 18 to 1D hex)

9. Power fail (IPL 1E hex)

10. Software interrupt invoked by MTPR #SIRR (IPL 01 to 0F hex)

11. AST delivery when REI restores a PSL with mode greater than or equal to ASTLVL (see chapter 7) (IPL 02)

Each device controller has a separate set of interrupt vector locations in the system control block (SCB). Thus interrupt service routines do not need to poll controllers in order to determine which controller interrupted. The vector address for each controller is fixed by hardware.

In order to reduce interrupt overhead, no memory mapping information is changed when an interrupt occurs. Thus the instructions, data, and contents of the interrupt vector for an interrupt service routine must be in the system address space or present in every process at the same address.

6.3.1 Urgent Interrupts -- Levels 18-1F (Hex)

The processor provides 8 priority levels for use by urgent conditions including serious errors (e.g., machine check) and power fail. Interrupts on these levels are initiated by the processor upon detection of certain conditions. Some of these conditions are not interrupts. For example, Machine Check is usually an exception but it runs at a high priority level on the interrupt stack.

Interrupt level 1E (hex) is reserved for power fail. Interrupt level 1F (hex) is reserved for those exceptions that must lock out all processing until handled. This includes the hardware and software "disasters" (machine check and kernel stack not valid). It might also be used to allow a kernel mode debugger to gain control on any exception.

6.3.2 Device Interrupts -- Levels 10-17 (Hex)

The processor provides 8 priority levels for use by peripheral devices. Any given implementation may or may not implement all 8 levels of interrupts. The minimal implementation is levels 14-17 (hex) that correspond to the UNIBUS levels BR4 to BR7 if the system has a UNIBUS.

6.3.3 Software Generated Interrupts -- Levels 01-0F (Hex)

6.3.3.1 Software Interrupt Summary Register - The processor provides 15 priority interrupt levels for use by software. Pending software interrupts are recorded in the Software Interrupt Summary Register (SISR). The SISR contains 1's in the bit positions corresponding to levels on which software interrupts are pending. All such levels, of course, must be lower than the current processor IPL, or the processor would have taken the requested interrupt.

```
 3                              1 1
 1                              6 5                           1 0
+-------------------------------+-----------------------------+-+
|                               | Pending Software Interrupts |M|
|             MBZ               |                             |B|
|                               |F E D C B A 9 8 7 6 5 4 3 2 1|Z|
+-------------------------------+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Software Interrupt Summary Register

The SISR is a read/write privileged register accessible only to privileged software (see Chapter 9). At bootstrap time, the contents of SISR is cleared. The mechanism for accessing it is:

MFPR #SISR,dst	Reads the software interrupt summary register.
MTPR src,#SISR	Loads it, but this is not the normal way of making software interrupt requests. It is useful for clearing the software interrupt system, and for reloading its state after a power fail, for example.

6.3.3.2 Software Interrupt Request Register - The software interrupt request register (SIRR) is a write-only four bit privileged register used for making software interrupt requests.

```
 3
 1                                                   4 3     0
+---------------------------------------------------+-------+
|                      ignored                      |request|
+---------------------------------------------------+-------+
```

Software Interrupt Request Register

Executing MTPR src,#SIRR requests an interrupt at the level specified by src<3:0>. Once a software interrupt request is made, it will be cleared by the hardware when the interrupt is taken. If src<3:0> is greater than the current IPL, the interrupt occurs before execution of the following instruction. If src<3:0> is less than or equal to the current IPL, the interrupt will be deferred until the IPL is lowered to less than src<3:0> and that there is no higher interrupt level pending. This lowering of IPL is by either REI or by MTPR x,#IPL. If src<3:0> is 0, no interrupt will occur.

Note that no indication is given if there is already a request at the selected level. Therefore, the service routine must not assume that there is a one-to-one correspondence of interrupts generated and requests made. A valid protocol for generating such a correspondence is:

1. The requester uses INSQUE to place a control block describing the request onto a queue for the service routine.

2. The requester uses MTPR src,#SIRR to request an interrupt at the appropriate level.

3. The service routine uses REMQUE to remove a control block from the queue of service requests. If REMQUE returns failure (nothing in the queue), the service routine exits with REI.

4. If REMQUE returns success (an item was removed from the queue), the service routine performs the service and returns to step 3 to look for other requests.

6.3.4 Interrupt Priority Level Register

Writing to the IPL with the MTPR instruction will load the processor priority field in the Program Status Longword (PSL), that is, PSL<20:16> is loaded from IPL<4:0>. Reading from IPL with the MFPR instruction will read the processor priority field from the PSL. On writing IPL bits <31:5> are ignored, on reading IPL bits <31:5> are returned zero.

```
 3
 1
                                                           5 4       0
+----------------------------------------------------------+----------+
|                    ignored; returns 0                    |PSL<20:16>|
+----------------------------------------------------------+----------+
```

Interrupt Priority Level Register

At bootstrap time, IPL is initialized to 31 (1F, hex).

Interrupt service routines must follow the discipline of not lowering IPL below their initial level. If they do, an interrupt at an intermediate level could cause the stack nesting to be improper. This would result in REI faulting . Actually, a service routine could lower the IPL if it ensures that no intermediate levels could interrupt, however this is probably unreliable code.

6.3.5 Interrupt Example

As an example, assume the processor is running in response to an interrupt at IPL5, it then sets IPL to 8, and then posts software requests at IPL3, IPL7, and IPL9. Then a device interrupt arrives at IPL11 (hex). Finally IPL is set back to IPL5. The sequence of execution is:

event	state after event contents of IPL (hex)	SISR (hex)	IPL in PSL on stack
(initial)	5	Ø	Ø
MTPR #8,#IPL	8	Ø	Ø
MTPR #3,#SIRR	8	8	Ø
MTPR #7,#SIRR	8	88	Ø
MTPR #9,#SIRR interrupts to	9	88	8,Ø
device interrupts to	11	88	9,8,Ø
device service routine REI	9	88	8,Ø
IPL9 service routine REI	8	88	Ø
MTPR #5,#IPL changes IPL to 5 and the request for 7 is granted immediately	7	8	5,Ø
IPL7 service routine REI	5	8	Ø
initial IPL5 service routine REI back to IPLØ and the request for 3 is granted immediately	3	Ø	Ø
IPL3 service routine REI	Ø	Ø	--

6.4 EXCEPTIONS

Exceptions can be grouped into six classes:

1. Arithmetic traps/faults
2. Memory management exceptions
3. Exceptions detected during operand reference
4. Exceptions occuring as a consequence of an instruction
5. Tracing
6. Serious system failures

6.4.1 Arithmetic Traps/Faults

This section contains the descriptions of the exceptions that occur as the result of performing an arithmetic or conversion operation. They are mutually exclusive and all are assigned the same vector in the SCB, and hence the same signal "reason" code. Each of them indicates that an exception had occurred during the last instruction and that the instruction has been completed (trap) or backed up (fault). An appropriate distinguishing code is pushed on the stack as a longword:

```
+-------------------------------------------------------------+
|                         type code                            | :(SP)
+-------------------------------------------------------------+
|             PC of next instruction to execute*              |
+-------------------------------------------------------------+
|                            PSL                              |
+-------------------------------------------------------------+
```

*same as the instruction causing exception in case of fault

type code (hex)	exception type	software mnemonic
	TRAPS	
1	integer overflow	SRM$K_INT_OVF_T
2	integer divide by zero	SRM$K_INT_DIV_T
3	floating overflow	SRM$K_FLT_OVF_T
4	floating/decimal divide by zero	SRM$K_FLT_DIV_T
5	floating underflow	SRM$K_FLT_UND_T
6	decimal overflow	SRM$K_DEC_OVF_T
7	subscript range	SRM$K_SUB_RNG_T
	FAULTS	
8	floating overflow	SRM$K_FLT_OVF_F
9	floating divide by zero	SRM$K_FLT_DIV_F
A	floating underflow	SRM$K_FLT_UND_F

6.4.1.1 Integer Overflow Trap - An integer overflow trap is an exception that indicates that the last instruction executed had an integer overflow setting the V condition code and that integer overflow was enabled (IV set). The result stored is the low-order part of the correct result. N and Z are set according to the stored result. The type code pushed on the stack is 1 (SRM$K_INT_OVF_T). Note that the instructions RET, REI, REMQUE, REMQHI, REMQTI, MOVTUC, and BISPSW do not cause overflow even if they set V. Also note that the EMODx floating point instructions can cause integer overflow.

6.4.1.2 Integer Divide By Zero Trap - An integer divide by zero trap is an exception that indicates that the last instruction executed had an integer zero divisor. The result stored is equal to the dividend and condition code V is set. The type code pushed on the stack is 2 (SRM$K_INT_DIV_T).

6.4.1.3 Floating Overflow Trap - A floating overflow trap is an exception that indicates that the last instruction executed resulted in an exponent greater than the largest representable exponent for the data type after normalization and rounding. The result stored contains a one in the sign and zeros in the exponent and fraction fields. This is a reserved operand, and will cause a reserved operand fault if used in a subsequent floating point instruction. The N and V condition code bits are set and Z and C are cleared. The type code pushed on the stack is 3 (SRM$K_FLT_OVF_T).

6.4.1.4 Divide By Zero Trap - Floating or Decimal String - A floating divide by zero trap is an exception that indicates that the last instruction executed had a floating zero divisor. The result stored is the reserved operand, as described above for floating overflow trap, and the condition codes are set as in floating overflow.

A decimal string divide by zero trap is an exception that indicates that the last instruction executed had a decimal string zero divisor. The destination, RØ through R5, and condition codes are UNPREDICTABLE. The zero divisor can be either +Ø or -Ø.

The type code pushed on the stack for both types of divide by zero is 4 (SRM$K_FLT_DIV_T).

6.4.1.5 Floating Underflow Trap - A floating underflow trap is an exception that indicates that the last instruction executed resulted in an exponent less than the smallest representable exponent for the data type after normalization and rounding and that floating underflow was enabled (FU set). The result stored is zero. Except for POLYx the N, V, and C condition codes are cleared and Z is set. In POLYx, the trap occurs on completion of the instruction, which may be many operations after the underflow. The condition codes are set on the final result in POLYx. The type code pushed on the stack is 5 (SRM K_FLT_UND_T).

6.4.1.6 Decimal String Overflow Trap - A decimal string overflow trap is an exception that indicates that the last instruction executed had a decimal string result too large for the destination string provided and that decimal overflow was enabled (DV set). The V condition code is always set. Refer to the individual instruction descriptions in Chapter 4 for the value of the result and of the condition codes. The type code pushed on the stack is 6 (SRM$K_DEC_OVF_T).

6.4.1.7 Subscript Range Trap - A subscript range trap is an exception that indicates that the last instruction was an INDEX instruction with a subscript operand that failed the range check. The value of the subscript operand is lower than the low operand or greater than the high operand. The result is stored in indexout, and the condition codes are set as if the subscript were within range. The type code pushed on the stack is 7 (SRM$K_SUB_RNG_T).

6.4.1.8 Floating Overflow Fault - A floating overflow fault is an exception that indicates that the last instruction executed resulted in an exponent greater than the largest representable exponent for the data type after normalization and rounding. The destination was unaffected and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with FPD set (see Chapter 4 for details). The type code pushed on the stack is 8 (SRM$K_FLT_OVF_F).

6.4.1.9 Divide By Zero Floating Fault - A floating divide by zero fault is an exception that indicates that the last instruction executed had a floating zero divisor. The quotient operand was unaffected and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. The type code pushed on the stack is 9 (SRM$K_FLT_DIV_F).

6.4.1.10 Floating Underflow Fault - A floating underflow fault is an exception that indicates that the last instruction executed resulted in an exponent less than the smallest representable exponent for the data type after normalization and rounding and that floating underflow was enabled (FU set). The destination operand is unaffected. The saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with FPD set (see Chapter 4 for details). The type code pushed on the stack is 10 (SRM K_FLT_UND_F).

6.4.2 Memory Management Exceptions

6.4.2.1 Access Control Violation Fault - An access control violation fault is an exception indicating that the process attempted a reference not allowed at the access mode at which the process was operating. See Chapter 5, Memory Management, for a description of the information pushed on the stack as parameters. Software may restart the process after changing the address translation information.

6.4.2.2 Translation Not Valid Fault - A translation not valid fault is an exception indicating that the process attempted a reference to a page for which the valid bit in the page table was not set. See Chapter 5, Memory Management, for a description of the information pushed on the stack as parameters. Note that if a process attempts to reference a page for which the page table entry specifies both Not Valid and Access Violation, an Access Control Violation Fault occurs.

6.4.3 Exceptions Detected During Operand Reference

6.4.3.1 Reserved Addressing Mode Fault - A reserved addressing mode fault is an exception indicating that an operand specifier attempted to use an addressing mode that is not allowed in the situation in which it occurred. No parameters are pushed.

The situations in which each specifier type is reserved are:

SPECIFIER	RESERVED SITUATION
Short Literal	Modify, destination, address source, or within index mode.
Register	Address source or within index mode.
Index Mode	Within index mode, or with PC as index.

See Chapter 3 for combinations of addressing modes and registers that cause UNPREDICTABLE results. The VAX-11/780 processor also faults on PC, @PC, and -(PC).

6.4.3.2 Reserved Operand Exception - A reserved operand exception is an exception indicating that an operand accessed has a format reserved for future use by DIGITAL. No parameters are pushed. This exception always backs up the PC to point to the opcode. The exception service routine may determine the type of operand by examining the opcode using the stored PC. Note that only the changes made by instruction fetch and because of operand specifier evaluation may be restored. Therefore, some instructions are not restartable. These exceptions are labelled as ABORTs rather than FAULTs. The PC is always restored properly unless the instruction attempted to modify it in a manner that results in UNPREDICTABLE results. The PSL other than FPD and TP is not changed except for the conditon codes, which are UNPREDICTABLE.

The reserved operand exceptions are caused by:

1. A floating point number that has the sign bit set and the exponent zero except in the POLY table (FAULT)

2. A floating point number that has the sign bit set and the exponent zero in the POLY table (FAULT; see chapter 4 for restartability)

3. POLY degree too large (FAULT)

4. Decimal string too long (ABORT)

5. Invalid digit in CVTTP, CVTSP (ABORT)

6. Bit field too wide (FAULT)

7. Invalid combination of bits in PSL restored by REI (FAULT)

8. Reserved pattern operator in EDITPC (FAULT; see Chapter 4 for restartability)

9. Incorrect source string length at completion of EDITPC (ABORT)

10. Invalid combination of bits in PSW/MASK longword during RET (FAULT)

11. Invalid combination of bits in BISPSW/BICPSW (FAULT)

12. Invalid CALLx entry mask (FAULT)

13. Invalid register number in MFPR or MTPR (FAULT)

14. Invalid combinations in PCB loaded by LDPCTX (ABORT)

15. Unaligned operand in ADAWI (FAULT)

16. Invalid register contents in MTPR instructions to some registers for some implementations (FAULT):

```
SISR<31:16>'SISR<0> NEQU 0
P0BR<1:0> NEQU 0
P0BR LSSU 2**31
P0BR GTRU 2**31+2**30-1
P1BR<1:0> NEQU 0
P1BR LSSU 2**31-2**23
P1BR GTRU 2**31+2**30-2**23-1
P0LR<31:27>'P0LR<23:22> NEQU 0
P1LR<30:22> NEQU 0
ASTLVL<2:0> GTRU 4
```

17. Invalid operand addresses in INSQHI, INSQTI, REMQHI, or REMQTI (FAULT)

6.4.4 Exceptions Occurring As The Consequence Of An Instruction

6.4.4.1 Opcode Reserved To DIGITAL fault - An opcode reserved to DIGITAL fault occurs when the processor encounters an opcode that is not specifically defined, or that requires higher privileges than the current mode. No parameters are pushed. Opcode FFFF (hex) will always fault.

6.4.4.2 Opcode Reserved To Customers (and CSS) Fault - An opcode reserved to customers fault is an exception that occurs when an opcode reserved to the customers or DIGITAL's Computer Special Systems group is executed. The operation is identical to the opcode reserved to DIGITAL fault except that the event is caused by a different set of opcodes, and faults through a different vector. All opcodes reserved to customers (and CSS) start with FC (hex), which is the XFC instruction. If the special instruction needs to generate a unique exception, one of the reserved to CSS/Customer vectors should be used. An example might be an unrecognized second byte of the instruction.

6.4.4.3 Compatibility Mode Exception - A compatibility mode exception is an exception that occurs when the processor is in compatibility mode. A longword of information is pushed on the stack, which contains a code as follows:

Ø	reserved opcode	FAULT
1	BPT	FAULT
2	IOT	FAULT
3	EMT	FAULT
4	TRAP	FAULT
5	illegal instruction	FAULT
6	odd address	ABORT

All other exceptions in compatibility mode occur to the regular VAX-11 vector, e.g., Access Control Violation, Translation Not Valid, Memory Error, and Machine Check Abort. See chapter 1Ø, Compatibility Mode.

6.4.4.4 Breakpoint Fault - A breakpoint fault is an exception that occurs when the breakpoint instruction (BPT) is executed. No parameters are pushed.

To proceed from a breakpoint, a debugger or tracing program typically restores the original contents of the location containing the BPT, sets T in the PSL saved by the BPT fault, and resumes. When the breakpointed instruction completes, a trace exception will occur (see section on tracing). At this point, the tracing program can again re-insert the BPT instruction, restore T to its original state (usually clear), and resume. Note that if both tracing and breakpointing are in progress (i.e., if PSL<T> was set at the time of the BPT), then on the trace exception both the BPT restoration and a normal trace exception should be processed by the trace handler.

6.4.5 Tracing

A trace is an exception that occurs between instructions when trace is enabled. Tracing is used for tracing programs, for performance evaluation, or debugging purposes. It is designed so that one and only one trace exception occurs before the execution of each traced instruction. The saved PC on a trace is the address of the next instruction that would normally be executed. If a trace fault and a memory management fault (or an odd address abort during a compatibility mode instruction fetch) occur simultaneously, the order in which the exceptions are taken is UNPREDICTABLE. The trace fault for an instruction takes precedence over all other exceptions.

In order to ensure that exactly one trace occurs per instruction despite other traps and faults, the PSL contains two bits, trace enable (T) and trace pending (TP). If only one bit were used then the occurrence of an interrupt at the end of an instruction would either produce zero or two traces, depending on the design. Instead of the PSL<T> bit being defined to produce a trap after any other traps or aborts at the end of an instruction, the trap effect is implemented by copying PSL<T> to a second bit (PSL<TP>) that is actually used to generate the exception. PSL<TP> generates a fault before any other processing at the start of the next instruction.

The rules of operation for trace are:

1. At the beginning of an instruction, if TP is set then a trace fault is taken after clearing TP.

2. TP is loaded with the value of T.

3. If the instruction faults or an interrupt is serviced, PSL<TP> is cleared before the PSL is pushed. The pushed PC is set to the start of the faulting or interrupted instruction. Instruction execution is resumed at Step 1.

4. If the instruction aborts or takes an arithmetic trap, PSL<TP> is not changed before the PSL is pushed.

5. If an interrupt is serviced after instruction completion and arithmetic traps but before tracing is checked for at the start of the next instruction, then PSL<TP> is not changed before the PSL is pushed.

The routine entered by a CHMx is not traced because CHMx clears T and TP in the new PSL. However, if T was set at the beginning of CHMx the saved PSL will have both T and TP set. Trace faults resume with the instruction following the REI in the routine entered by the CHMx. An instruction following an REI will fault either if T was set when the REI was executed or if TP in the saved PSL is set; in both cases TP is set after the REI. Note that a trace fault that occurs for an instruction following an REI that sets TP will be taken with the new PSL. Thus, special care must be taken if exception or interrupt routines are traced. If the T bit is set by a BISPSW instruction, trace faults begin with the second instruction after the BISPSW.

In addition, the CALLx instructions save a clear T, although T in the PSL is unchanged. This is done so that a debugger or trace program proceeding from a BPT fault does not get a spurious trace from the RET that matches the CALL.

The detection of reserved instruction faults occurs after the trace fault. The detection of interrupts and other exceptions can occur during instruction execution. In this case, TP is cleared before the exception or interrupt is initiated. The entire PSL (including T and TP) is automatically saved on interrupt or exception initiation and is restored at the end with an REI. This makes interrupts and benign exceptions totally transparent to the executing program.

6.4.5.1 Trace Instruction Summary - The following table shows all of the cases of T enabled at the beginning of the instruction, enabled at the end of the instruction, and TP set in the popped PSW or PSL for ordinary instructions (XXX), CHMx...REI, interrupt or exception...REI, CALLx, RETURN, CHMx, REI, BISPSW, and BICPSW:

Trace exception

	enabled at beg (T)	enabled at end (T)	TP bit at end (TP)	
XXX	N	N	N	
	Y	Y	Y	
CHMx...REI	N	N	N	
	Y	Y	Y	
interrupt or exception...REI	N	N	N	
	Y	Y	Y	
CALLx	N	N	N	
	Y	Y	Y	(pushed PSW<T> clear)
RET	N	N*	N	
	N	Y*	N	(no fault before next instruction)
	Y	N*	Y	
	Y	Y*	Y	
CHMx	N	N	N	(pushed PSL<TP> clear)
	Y	N	N	(pushed PSL<TP> set)
REI (if PSL<TP>=0 on stack)	N	N*	N	
	N	Y*	N	
	Y	N*	Y	
	Y	Y*	Y	
REI (if PSL<TP>=1 on stack)	N	N*	Y	
	N	Y*	Y	
	Y	N*	Y	
	Y	Y*	Y	
BISPSW	N	Y	N	
	Y	Y	Y	
BICPSW	N	N	N	
	Y	N	Y	
interrupt or exception	N	N	N	(pushed PSL<TP> clear)
	Y	N	N	(pushed PSL<TP> depends on above description)

* = depends on PSW<T> popped from stack

6.4.5.2 Using Trace - Routines using the trace facility are termed trace handlers. They should observe the following conventions and restrictions:

1. When the trace handler performs its REI back to the traced program, it should always force the T bit on in the PSL that will be restored. This defends against programs clearing T via RET, REI, or BICPSW.

2. The trace handler should never examine or alter the TP bit when continuing tracing. The hardware flows ensure that this bit is maintained correctly to continue tracing.

3. When tracing is to be ended, both T and TP should be cleared. This ensures that no further traces will occur.

4. Tracing a service routine that completes with an REI will give a trace in the restored mode after the REI. If the program being restored to was also being traced, only one trace exception is generated.

5. If a routine entered by a CALLx instruction is executed at full speed by turning off T, then trace control can be regained by setting T in the PSW in its call frame. Tracing will resume after the instruction following the RET.

6. Tracing is disabled for routines entered by a CHMx instruction or any exception. Thus, if a CHMx or exception service routine is to be traced, a breakpoint instruction must be placed at its entry point. If such a routine is recursive, breakpointing will catch each recursion only if the breakpoint is not on the CHMx or instruction with the exception.

7. If it is desired to allow multiple trace handlers, all handlers should preserve T when turning on and off trace. They also would have to simulate traced code that alters or reads T.

6.4.6 Serious System Failures

6.4.6.1 Kernel Stack Not Valid Abort - Kernel stack not valid abort is an exception that indicates that the Kernel stack was not valid while the processor was pushing information onto the Kernel stack during the initiation of an exception or interrupt. Usually this is an indication of a stack overflow or other executive software error. The attempted exception is transformed into an abort that uses the interrupt stack. No extra information is pushed on the interrupt stack in addition to PSL and PC. IPL is raised to 1F (hex). Software may abort the process without aborting the system. However, because of the lost information, the process cannot be continued. If the Kernel Stack is not valid during the normal execution of an instruction (including CHMK or REI), the normal memory management fault is initiated. If the exception vector <1:Ø> for Kernel Stack Not Valid is 3, the behavior of the processor is UNDEFINED (see section on SCB vectors).

6.4.6.2 Interrupt Stack Not Valid Halt - An interrupt stack not valid halt is an exception that indicates that the interrupt stack was not valid or that a memory error occurred while the processor was pushing information onto the interrupt stack during the initiation of an exception or interrupt. No further interrupt requests are acknowledged on this processor. The processor leaves the PC, the PSL, and the reason for the halt in registers so that it is available to a debugger, the normal bootstrap routine, or an optional watch dog bootstrap routine. A watch dog bootstrap can cause the processor to leave the halted state.

6.4.6.3 Machine Check Exception - A machine check exception indicates that the processor detected an internal error in itself. As usual for exceptions, this exception is taken independent of IPL. IPL is raised to 1F (hex) only if vector<1:Ø> is 1..

Implementation specific information is pushed on the stack as longwords. The processor specifies the number of bytes pushed by placing the number of bytes pushed as the last longword pushed. (Ø if none, 4 if one, ...). This count excludes the PC, PSL, and count longwords. Software can decide, on the basis of the information presented, whether to abort the current process if the machine check came from the process. Machine check includes uncorrected bus and memory errors anywhere, and any other processor-detected errors. Some processor errors cannot ensure the state of the machine at all. For such errors, the state will be preserved on a "best effort" basis. If the exception vector <1:Ø> for machine check is 3, the behavior of the processor is UNDEFINED (see section on SCB vectors).

6.5 SERIALIZATION OF NOTIFICATION OF MULTIPLE EVENTS

The interaction between arithmetic traps, tracing, other exceptions, and multiple interrupts is complex. In order to ensure consistent and useful implementations, it is necessary to understand this interraction at a detailed level. As an example, if an instruction is started with T=1 and TP=Ø, it gets an arithmetic trap, and an interrupt request is recognized, the following sequence occurs:

1. The instruction finishes, storing all its results. PSL<TP> is set at the end of this instruction since PSL<T> was set at the beginning.

2. The overflow trap sequence is initiated, pushing the PC and PSL (with TP=1), loading a new PC from the vector, and creating a new PSL.

3. The interrupt sequence is initiated, pushing the PC and PSL appropriate to the overflow trap service routine, loading a new PC from the vector, and creating a new PSL.

4. If a higher priority interrupt is noticed, the first instruction of the interrupt service routine is not executed. Instead, the PC and PSL appropriate to that routine are saved as part of initiating the new interrupt. The original interrupt service routine will then be executed when the higher priority routine terminates via REI.

5. The interrupt service routine runs, and exits with REI.

6. The overflow trap service routine runs, and exits with REI, which sets PSL<TP> since the saved PSL<TP> was set.

7. The trace fault occurs, again pushing PC and PSL but this time with TP=Ø.

8. Trace service routine runs, and exits with REI.

9. The next instruction is executed.

This is accomplished by the following operation between instructions:

```
        !here at completion of instruction including
        !  at end of REI from an exception or interrupt routine

   1$:  {possibly take interrupts or console halt};
        !PSL<TP> is not modified before PSL is saved

        if PSL<TP> EQLU 1 then          !if trace pending, take trace fault.
                begin                   !Trace fault takes precedence
                PSL<TP> <- 0;           !over other exceptions.
                {initiate trace fault};
                end;

        {possibly take interrupts or console halt};
        !PSL<TP> is not modified before PSL is saved

        PSL<TP> <- PSL<T>;      !if trace enable, set trace pending

        {go start instruction execution};
        !Reserved instruction faults are taken here
        !FPD is tested here, thus TP takes
        ! precedence over FPD if both are set.
        if {instruction faults} OR {an interrupt or console halt
            is taken before end of instruction} then
                begin
                {back up PC to start of opcode};
                {either set PSL<FPD> or back up all general
                 register side effects};
                PSL<TP> <- 0;
                {initiate exception or interrupt};

        if {arith trap needed and no other abort
                or trap} then {initiate arith trap};

                end;

                !note: all instructions end by flowing
                ! through 1$, thus the REI from a service
                ! routine will return to 1$
```

6.6 SYSTEM CONTROL BLOCK (SCB)

The System Control Block is a page containing the vectors by which exceptions and interrupts are dispatched to the appropriate service routines.

6.6.1 System Control Block Base (SCBB)

The SCBB is a privileged register containing the physical address of the System Control Block, which must be page-aligned.

```
 3 3 2
 1 0 9                                      9 8                 0
+---+----------------------------------------+------------------+
|MBZ|      Physical page address of SCB      |        MBZ       |
+---+----------------------------------------+------------------+
```

System Control Block Base

At bootstrap time, the contents of SCBB is UNPREDICTABLE. The actual length is implementation dependent because it represents a physical address.

6.6.2 Vectors

A vector is a longword in the SCB that is examined by the processor when an exception or interrupt occurs, to determine how to service the event.

Separate vectors are defined for each interrupting device controller and each class of exceptions. Each vector is interpreted as follows by the hardware. Bits 1:0 contain a code interpreted:

0. Service this event on the kernel stack unless already running on the interrupt stack, in which case service on the interrupt stack.

1. Service this event on the interrupt stack. If this event is an exception, the IPL is raised to 1F (hex).

2. Service this event in writable control store, passing bits 15:2 to the installation-specific microcode there. If writable control store does not exist or is not loaded, the operation is UNDEFINED. On the VAX-11/780 processor, the operation in this case is a HALT.

3. Operation UNDEFINED. Reserved to DIGITAL. On the VAX-11/780 processor, the operation is a HALT.

For codes 0 and 1, bits 31:2 contain the virtual address of the service routine, which must begin on a longword boundary and will ordinarily be in the system space. CHMx is serviced on the stack selected by the new mode. Bits <1:0> in the CHMx vectors must be zero or the operation is UNDEFINED. On the VAX-11/780 processor, these bits are ignored in the CHMx vectors.

System Control Block (exception and interrupt vectors)

Vector (hex)	Name	Type	Number of Params	Notes
ØØ	Unused			Reserved to DIGITAL.
Ø4	Machine Check	Abort/ Fault/ Trap	*	Processor-and error-specific information is pushed on the stack, if possible. Restartability is processor specific. If vector<1:Ø> is 1, IPL is raised to 1F(hex) and the interrupt stack is used (i.e. IS <- 1).. * -- the number of bytes of parameters is pushed on the stack and is implementation dependent.
Ø8	Kernel Stack Not Valid	Abort	Ø	Serviced on the interrupt stack (i.e. IS <- 1). IPL is raised to 1F (hex).
ØC	Power Fail	Interrupt	Ø	IPL is raised to 1E (hex).
1Ø	Reserved/Privileged Instruction	Fault	Ø	Opcodes reserved to DIGITAL and privileged instructions.
14	Customer Reserved Instruction	Fault	Ø	XFC instruction.
18	Reserved Operand	Fault/ Abort	Ø	Type depends on circumstances. See section on reserved operand exceptions.
1C	Reserved Addressing Mode	Fault	Ø	
2Ø	Access Control Violation	Fault	2	Virtual address causing fault is pushed onto stack. See chapter 5.

24	Translation Not Valid	Fault	2	Virtual address causing fault is pushed onto stack. See chapter 5.
28	Trace Pending (TP)	Fault	Ø	
2C	Breakpoint Instruction	Fault	Ø	
3Ø	Compatibility	Fault/ Abort	1	A type code is pushed onto the stack. See section on compatibility mode exceptions.
34	Arithmetic	Trap/ Fault	1	A type code is pushed onto the stack. See 6.4.
38-3C	Unused			Reserved to DIGITAL.
4Ø	CHMK	Trap	1	The operand word is sign extended and pushed onto the stack. Vector<1:Ø> MBZ.
44	CHME	Trap	1	The operand word is sign extended and pushed onto the stack. Vector<1:Ø> MBZ.
48	CHMS	Trap	1	The operand word is sign extended and pushed onto the stack. Vector<1:Ø> MBZ.
4C	CHMU	Trap	1	The operand word is sign extended and pushed onto the stack. Vector<1:Ø> MBZ.
5Ø	SBI SILO Compare	Interrupt	Ø	IPL is 19 (hex). VAX-11/78Ø only.
54	Corrected Memory Read Data	Interrupt	*	IPL is 1A (hex). Also used for Read Data Substitute on VAX-11/78Ø. Number of parameters is implementation dependent.
58	SBI Alert	Interrupt	Ø	IPL is 1B (hex). VAX-11/78Ø only.
5C	SBI Fault	Interrupt	Ø	IPL is 1C (hex).

			VAX-11/780 only.
60	Memory Write Timeout	Interrupt *	IPL is 1D (hex). Number of parameters is implementation dependent.
64-80	Unused		Reserved to DIGITAL.
84	Software Level 1	Interrupt 0	
88	Software Level 2	Interrupt 0	Ordinarily used for AST delivery.
8C	Software Level 3	Interrupt 0	Ordinarily used for Process Scheduling.
90-BC	Software Levels 4-F	Interrupt 0	
C0	Interval Timer	Interrupt 0	IPL is 18 (hex).
C4-DC	Unused		Reserved to DIGITAL
E0-EC	Unused		Reserved to CSS/Customers
F0	Console Storage Rec.	Interrupt 0	IPL is 17 (hex). VAX-11/750 only.
F4	Console Storage Trans.	Interrupt 0	IPL is 17 (hex). VAX-11/750 only.
F8	Console Terminal Rec.	Interrupt 0	IPL is 14 (hex).
FC	Console Terminal Trans.	Interrupt 0	IPL is 14 (hex).
100-3FC	Device Vectors	Interrupt 0	

In the VAX-11/780 processor, only interrupt priority levels 14 to 17 (hex) are available to a NEXUS external to the CPU, and there is a limit of 16 such NEXUS. A NEXUS is a connection on the SBI, which is the internal interconnection structure. The NEXUS vectors are assigned as follows:

100-13C	IPL 14 (hex)	NEXUS 0-15
140-17C	IPL 15 (hex)	NEXUS 0-15
180-1BC	IPL 16 (hex)	NEXUS 0-15
1C0-1FC	IPL 17 (hex)	NEXUS 0-15

In the VAX-11/750 processor, UNIBUS devices interrupt the processor directly. The vector is determined by adding 200 (hex) to the vector supplied by the device. Only SCB vectors in the range 200 to 3FC (hex) are allowed. Interrupt priority levels 14 to 17 (hex) correspond to UNIBUS levels BR4 to BR7.

6.7 STACKS

At any time, the processor is either in a process context (IS=Ø) in one of four modes (kernel, exec, super, user), or in the system-wide interrupt service context (IS=1) that operates with kernel privileges. There is a stack pointer associated with each of these five states, and any time the processor changes from one of these states to another, SP (R14) is stored in the process context stack pointer for the old state and loaded from that for the new state. The process context stack pointers (KSP=kernel, ESP=exec, SSP=super, USP=user) are allocated in the PCB (see Chapter 7), although some hardware implementations may keep them in privileged registers. The interrupt stack pointer (ISP) is in a privileged register.

Operating system design must choose a priority level that is the boundary between kernel and interrupt stack use. The SCB interrupt vectors must be set such that interrupts to levels above this boundary run on the interrupt stack (vector<1:Ø> = 1) and interrupts below this boundary run on the kernel stack (vector<1:Ø> = Ø). Typically, AST delivery (IPL 2) is on the kernel stack and all higher levels are on the interrupt stack.

6.7.1 Stack Residency

The USER, SUPER, and EXEC stacks do not need to be resident. The kernel can bring in or allocate process stack pages as Address Translation Not Valid faults occur. However, the kernel stack for the current process, and the interrupt stack (which is process-independent) must be resident and accessible. Translation Not Valid and Access Control Violation faults occurring on references to either of these stacks are regarded as serious system failures, from which recovery is not possible.

If either of these faults occurs on a reference to the kernel stack, the processor aborts the current sequence and initiates Kernel Stack Not Valid abort on hardware level 1F (hex). If either of these faults occurs on a reference to the interrupt stack, the processor halts. Note that this does not mean that every possible reference is checked, but rather that the processor will not loop on these conditions.

It is not necessary that the kernel stack for processes other than the current one be resident, but it must be resident before a process is selected to run by the software's process dispatcher. Further, any mechanism that uses Translation Not Valid or Access Control Violation faults to gather process statistics, for instance, must exercise care not to invalidate kernel stack pages.

6.7.2 Stack Alignment

Except on CALLx instructions, the hardware makes no attempt to align the stacks. For best performance on all processors, the software should align the stack on a longword boundary and allocate the stack in longword increments. The convert byte to long (CVTBL and MOVZBL), convert word to long (CVTWL and MOVZWL), convert long to byte (CVTLB), and convert long to word (CVTLW) instructions are recommended for pushing bytes and words on the stack and popping them off in order to keep it longword aligned.

6.7.3 Stack Status Bits

The interrupt stack bit (IS) and current mode bits in the privileged Processor Status Longword (PSL) specify which of the five stack pointers is currently in use as follows:

IS	MODE	REGISTER
1	Ø	ISP
Ø	Ø	KSP
Ø	1	ESP
Ø	2	SSP
Ø	3	USP

The processor does not allow current mode to be non-zero when IS=1. This is achieved by clearing the mode bits when taking an interrupt or exception, and by causing reserved operand fault if REI attempts to load a PSL in which both IS and mode are non-zero.

The stack to be used for an interrupt or exception is selected by the current PSL<IS> and bits <1:Ø> of the vector for the event as follows:

```
                    vector<1:Ø>
                      ØØ    Ø1
                   +-----+-----+
                Ø  | KSP | ISP |
      PSL<IS>      +-----+-----+
                1  | ISP | ISP |
                   +-----+-----+
```

Values 1Ø (binary) and 11 (binary) of the vector<1:Ø> are used for other purposes. Refer to section on SCB vectors for details.

6.7.4 Accessing Stack Registers

Reference to SP (the stack pointer) in the general registers will access one of five possible architecturally defined stack pointers; the user, supervisor, executive, kernel, or interrupt stack pointer, depending on the values of the current mode and IS bits in the PSL. Some processors might implement these five stack pointers as five internal processor registers. On these processors, software can access any of the five stack pointers not currently selected by the current mode and IS bits in the PSL via the MTPR and MFPR instructions. Results are correct even if the stack pointer specified by the current mode and IS bits in the PSL is referenced in the processor register space by an MTPR or MFPR instruction. If the process stack pointers are implemented as registers, then these instructions are the only method for accessing the stack pointers of the current process. If the process stack pointers are kept only in the PCB, MTPR and MFPR of these registers might not access the PCB. See Chapter 9 for conventions to be followed when referencing per-process registers that are also in the processor register space.

The internal processor register numbers were chosen to be the same as PSL<26:24> (see Chapter 9). The previous stack pointer is the same as PSL<23:22> unless PSL<IS> is set. If PSL<IS> is set, the previous mode cannot be determined from the PSL since interrupts always clear PSL<23:22>. At bootstrap time, the contents of all stack pointers are UNPREDICTABLE.

6.8 INITIATE EXCEPTION OR INTERRUPT

Condition Codes (if vector<1:0> code is 0 or 1):

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

interrupt stack not valid
kernel stack not valid

Description:

The handling is determined by the contents of a longword vector in the system control block which is indexed by the exception or interrupt being processed. If the processor is not executing on the interrupt stack, then the current stack pointer is saved and the new stack pointer is fetched. The old PSL is pushed onto the new stack. The PC is backed up (unless this is an interrupt between instructions or a trap) and is pushed onto the new stack. The PSL is initialized to a canonical state. IPL is changed if this is an interrupt or if it is an exception with vector<1:0> code 1. Any parameters are pushed. Except for interrupts, the previous mode in the new PSL is set to the old value of the current mode. Finally, the PC is changed to point to the longword indicated by the vector<31:2>.

Notes:

1. Interrupts are disabled during this sequence.

2. If the vector<1:0> code is invalid, the behavior is UNDEFINED.

3. On an abort, the saved condition codes are UNPREDICTABLE. On a fault or interrupt, the saved condition codes are UNPREDICTABLE; they are only saved to the extent necessary to ensure correct completion of the instruction when resumed. On an abort or fault or interrupt that sets FPD, the general registers except PC, SP and FP are UNPREDICTABLE unless the instruction description specifies a setting. If FP is the destination in this case, then it is also UNPREDICTABLE. On a Kernel Stack Not Valid abort, both SP and FP are UNPREDICTABLE. In this case, UNPREDICTABLE means unspecified; upon REI the instruction behavior and results are predictable. This implies that processes stopped with FPD set cannot be resumed on processors of a different type or engineering change level.

4. If the processor gets an Access Control Violation or a Translation Not Valid condition while attempting to push information on the kernel stack, a Kernel Stack Not Valid abort is initiated and IPL is changed to 1F (hex). The additional

information, if any, associated with the original exception is lost. However PSL and PC are pushed on the interrupt stack with the same values as would have been pushed on the kernel stack.

5. If the processor gets an Access Control Violation or a Translation Not Valid condition while attempting to push information on the interrupt stack, the processor is halted and only the state of ISP, PC, and PSL is insured to be correct for subsequent analysis. The PSL and PC have the values that would have been pushed on the interrupt stack.

6. The value of PSL<TP> that is saved on the stack is as follows:

fault	clear
trace	clear
interrupt	clear (if FPD set) from PSL<TP> (if after traps, before trace)
abort	from PSL<TP>
trap	from PSL<TP>
CHMx	from PSL<TP>
BPT, XFC	clear
reserv.instr.	clear

7. The value of PC that is saved on the stack points to the following:

fault	instruction faulting
trace	next instruction to execute i.e. instruction at the beginning of which the trace fault was taken.
interrupt	instruction interrupted or next instruction to execute
abort	instruction aborting or detecting Kernel Stack Not Valid (not ensured on machine check)
trap	next instruction to execute
CHMx	next instruction to execute
BPT, XFC	BPT, XFC instruction
reserv.instr.	reserv.instr.

8. The non-interrupt stack pointers may be fetched and stored by hardware in either privileged registers or in their allocated slots in the PCB. Only LDPCTX and SVPCTX always fetch and store in the PCB, see Chapter 7. MFPR and MTPR always fetch and store the pointers whether in registers or the PCB.

6.9 RELATED INSTRUCTIONS

REI   Return from Exception or Interrupt

Format:

Opcode

Operation:

```
tmp1 <- (SP)+;   ! Pick up saved PC
tmp2 <- (SP)+;   ! and PSL

if {tmp2<CUR_MOD> LSSU PSL<CUR_MOD>} OR
   {tmp2<IS> EQLU 1 AND PSL<IS> EQLU Ø} OR
   {tmp2<IS> EQLU 1 AND tmp2<CUR_MOD> NEQU Ø} OR
   {tmp2<IS> EQLU 1 AND tmp2<IPL> EQLU Ø} OR
   {tmp2<IPL> GTRU Ø AND tmp2<CUR_MOD> NEQU Ø} OR
   {tmp2<PRV_MOD> LSSU tmp2<CUR_MOD>} OR
   {tmp2<IPL> GTRU PSL<IPL>} OR
   {tmp2<PSL_MBZ> NEQU Ø} then {reserved operand fault};
if {tmp2<CM> EQLU 1} AND
   {{tmp2<FPD,IS,DV,FU,IV> NEQU Ø} OR
    {tmp2<CUR_MOD> NEQU 3}} then {reserved operand fault};

if PSL<IS> EQLU 1 then ISP <- SP          !save old stack pointer
                  else PSL<CUR_MOD>_SP <- SP;
if PSL<TP> EQLU 1 then tmp2<TP> <- 1;     !TP <- TP or stack TP
PC <- tmp1;
PSL <- tmp2;
if PSL<IS> EQLU Ø then
        begin
        SP <- PSL<CUR_MOD>_SP;             !switch stack
        if PSL<CUR_MOD> GEQU ASTLVL       !check for AST delivery
                then {request interrupt at IPL 2};
        end;
{check for software interrupts};
{clear instruction look-ahead}
```

Condition Codes:

```
N <- saved PSL<3>;
Z <- saved PSL<2>;
V <- saved PSL<1>;
C <- saved PSL<Ø>;
```

Exceptions:

reserved operand

Opcodes:

Ø2	REI	Return from Exception or Interrupt

Description:

A longword is popped from the current stack and held in a temporary PC. A second longword is popped from the current stack and held in a temporary PSL. Validity of the popped PSL is checked. The current stack pointer is saved and a new stack pointer is selected according to the new PSL CUR_MOD and IS fields (see section on Stack Status Bits). The level of the highest privilege AST is checked against the current mode to see whether a pending AST can be delivered; refer to chapter 7. Execution resumes with the instruction being executed at the time of the exception or interrupt. Any instruction lookahead in the processor is reinitialized.

Notes:

1. The exception or interrupt service routine is responsible for restoring any registers saved and 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 non-interrupt stack pointers may be fetched and stored either in privileged registers or in their allocated slots in the PCB. Only LDPCTX and SVPCTX always fetch and store in the PCB (see Chapter 7). MFPR and MTPR always fetch and store the pointers whether in registers or the PCB.

CHM Change Mode

Purpose: request services of more privileged software

Format:

```
        opcode  code.rw
```

Operation:

```
        tmp1 <- {mode selected by opcode (K=Ø, E=1, S=2, U=3)};
        tmp2 <- MINU(tmp1, PSL<CUR_MOD>);        !maximize privilege
        tmp3 <- SEXT(code);
        if {PSL<IS> EQLU 1} then HALT;           !illegal from I stack

        PSL<CUR_MOD>_SP <- SP;                   !save old stack pointer
        tmp4 <- tmp2_SP;                         !get new stack pointer
        PROBEW (from tmp4-1 through tmp4-12 with mode=tmp2);    !check
                                                 ! new stack access
            if {access control violation} then
                {initiate access violation fault};
            if {translation not valid} then
                {initiate translation not valid fault};

        {initiate CHMx exception with new_mode=tmp2
                and parameter=tmp3
                using 4Ø+tmp1*4 (hex) as SCB offset
                using tmp4 as the new SP
                and not storing SP again};
```

Condition Codes:

```
        N <- Ø;
        Z <- Ø;
        V <- Ø;
        C <- Ø;
```

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 only increases privilege (i.e., decreases the access mode).

A change in mode also results in a change of stack pointers; the old pointer is saved, the new pointer is loaded. The PSL, PC, and code passed by the instruction are pushed onto the stack of the new mode. The saved PC addresses the instruction following the CHMx instruction. The code is sign extended. After execution, the new stack's appearance is:

```
+---------------------------------------------------------------+
|                     sign extended code                        | :(SP)
+---------------------------------------------------------------+
|                    PC of next instruction                     |
+---------------------------------------------------------------+
|                           old PSL                             |
+---------------------------------------------------------------+
```

The destination mode selected by the opcode is used to obtain a location from the System Control Block. This location addresses the CHMx dispatcher for the specified mode. If the vector<1:Ø> code NEQU Ø then the operation is UNDEFINED.

Notes:

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

2. The non-interrupt stack pointers may be fetched and stored either in privileged registers or in their allocated slots in the PCB. Only LDPCTX and SVPCTX always fetch and store in the PCB, see Chapter 7. 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
```

6.10 PROCESSOR STATE TRANSITION TABLE

FINAL STATE

\ INITIAL STATE	User IS=Ø IPL=Ø	Super IS=Ø IPL=Ø	Exec IS=Ø IPL=Ø	Kernel IS=Ø IPL=Ø	Kernel IS=Ø IPL>Ø	Kernel IS=1 IPL>Ø	Program Halt

User
IS=Ø
IPL=Ø
	CHMU
	REI
		CHMS	CHME	CHMK
				Excep(Ø)
					Inter(Ø)	Excep(1)
						Inter(1)
							impos-
							sible

Super
IS=Ø
IPL=Ø
	REI*	CHMU,S
		REI
			CHME	CHMK
				Excep(Ø)
					Inter(Ø)	Excep(1)
						Inter(1)
							impos-
							sible

Exec
IS=Ø
IPL=Ø
	REI*	REI*	CHMU,S,E
			REI
				CHMK
				Excep(Ø)
					Inter(Ø)	Excep(1)
						Inter(1)
							impos-
							sible

Kernel
IS=Ø
IPL=Ø
	REI*	REI*	REI*	CHMUSEK
				REI*
				Excep(Ø)
				MTPR IPL
				LDPCTX
					MTPR IPL
					Inter(Ø)
						SVPCTX
						Excep(1)
						Inter(1)
							HALT
							Instr.

Kernel
IS=Ø
IPL>Ø
	REI*	REI*	REI*	MTPR IPL
				REI*
					CHMUSEK
					REI*
					Excep(Ø)
					Inter(Ø)
					MTPR IPL
					LDPCTX
						SVPCTX
						Excep(1)
						Inter(1)
							HALT
							Instr.

Kernel IS=1 IPL>Ø	REI*	REI*	REI*	REI*	LDPCTX REI*	SVPCTX REI Excep Inter MTPR IPL	HALT Instr. CHMUSEK

Inter is Interrupt	(Ø) is vector<1:Ø> = Ø
Excep is Exception	(1) is vector<1:Ø> = 1

* Any REI that increases mode can cause an interrupt request at IPL 2 for AST delivery.

Processor State Transitions

CHAPTER 7
PROCESS STRUCTURE

21-May-80 -- Rev 5

7.1 PROCESS DEFINITION

A process is a single thread of execution. It is the basic schedulable entity that is executed by the processor. A process consists of an address space and both hardware and software context. The hardware context of a process is defined by a Process Control Block (PCB) that contains images of the 14 general purpose registers, the processor status longword (PSL), the program counter (PC), the 4 per-process stack pointers, the process virtual memory defined by the base and length registers PØBR, PØLR, P1BR, and P1LR and several minor control fields. In order for a process to execute, the majority of the PCB must be moved into the internal registers. While a process is executing, some of its hardware context is being updated in the internal registers. When a process is not being executed its hardware context is stored in a data structure termed the Process Control Block (PCB). Saving the contents of the privileged registers in the PCB of the currently executing process and then loading a new context from another PCB is termed context switching. Context switching occurs as one process after another is scheduled for execution.

7.2 PROCESS CONTEXT

7.2.1 Process Control Block Base (PCBB)

The process control block for the currently executing process is pointed to by the content of the Process Control Block Base (PCBB) register, an internal privileged register. Figure 7.1 depicts the Process Control Block Base.

```
 3 3 2                                                          2 1 0
 1 0 9
+---+-----------------------------------------------------------+---+
|MBZ|              physical longword address of PCB             |MBZ|
+---+-----------------------------------------------------------+---+

                              (read/write)

                 Process Control Block Base (PCBB) Register
```

At bootstrap time, the contents of PCBB is UNPREDICTABLE.

7.2.2 Process Control Block (PCB)

The process control block (PCB) contains all of the switchable process context collected into a compact form for ease of movement to and from the privileged internal registers. Although in any normal operating system there is additional software context for each process, the following description is limited to that portion of the PCB known to the hardware. Figure 7-2 depicts the PCB, whose contents are described in Table 7-1.

```
 31                                                              0
+-----------------------------------------------------------------+
|                              KSP                                |:PCB
+-----------------------------------------------------------------+
|                              ESP                                | +4
+-----------------------------------------------------------------+
|                              SSP                                | +8
+-----------------------------------------------------------------+
|                              USP                                | +12
+-----------------------------------------------------------------+
|                              RØ                                 | +16
+-----------------------------------------------------------------+
|                              R1                                 | +2Ø
+-----------------------------------------------------------------+
|                              R2                                 | +24
+-----------------------------------------------------------------+
|                              R3                                 | +28
+-----------------------------------------------------------------+
|                              R4                                 | +32
+-----------------------------------------------------------------+
|                              R5                                 | +36
+-----------------------------------------------------------------+
|                              R6                                 | +4Ø
+-----------------------------------------------------------------+
|                              R7                                 | +44
+-----------------------------------------------------------------+
|                              R8                                 | +48
+-----------------------------------------------------------------+
|                              R9                                 | +52
+-----------------------------------------------------------------+
|                              R1Ø                                | +56
+-----------------------------------------------------------------+
|                              R11                                | +6Ø
+-----------------------------------------------------------------+
|                            AP(R12)                              | +64
+-----------------------------------------------------------------+
|                            FP(R13)                              | +68
+-----------------------------------------------------------------+
|                              PC                                 | +72
+-----------------------------------------------------------------+
|                              PSL                               | +76
+-----------------------------------------------------------------+
|                              PØBR                               | +8Ø
+---------+-----+---+---------------------------------------------+
|         | AST |   |                                             | +84
|   MBZ   | LVL |MBZ|                   PØLR                      |
+---------+-----+---+---------------------------------------------+
|                              P1BR                               | +88
+-+-----------------+---------------------------------------------+
|P|                 |                                             | +92
|M|       MBZ       |                      P1LR                   |
|E|                 |                                             |
+-+-----------------+---------------------------------------------+
```

Figure 7-2 Process Control Block (PCB)

Table 7-1

Description of Process Control Block

Longword	Bits	Mnemonic	Description
Ø	<31:Ø>	KSP	Kernel Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is Ø and IS = Ø.
1	<31:Ø>	ESP	Executive Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 1.
2	<31:Ø>	SSP	Supervisor Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 2.
3	<31:Ø>	USP	User Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 3.
4-17	<31:Ø>	RØ-R11, AP,FP	General registers RØ through R11, AP, FP.
18	<31:Ø>	PC	Program Counter.
19	<31:Ø>	PSL	Program Status Longword.
2Ø	<31:Ø>	PØBR	Base register for page table describing process virtual addresses from Ø to 2**3Ø-1. See chapter 5.
21	<21:Ø>	PØLR	Length register for page table located by PØBR. Describes effective length of page table. See chapter 5.
21	<23:22>	MBZ	Must be zero.

21	<26:24>	ASTLVL	Contains access mode number (established by software) of the most privileged access mode for which an AST is pending. Controls the triggering of the AST delivery interrupt during REI instructions.

ASTLVL	Meaning
Ø	AST pending for access mode Ø (kernel)
1	AST pending for access mode 1 (executive)
2	AST pending for access mode 2 (supervisor)
3	AST pending for access mode 3 (user)
4	No pending AST
5-7	Reserved to DIGITAL

21	<31:27>	MBZ	Must be zero.
22	<31:Ø>	P1BR	Base register for page table describing process virtual addresses from 2**3Ø to 2**31-1. See chapter 5.
23	<21:Ø>	P1LR	Length register for page table located by P1BR. Describes effective length of page table. See chapter 5.
23	<3Ø:22>	MBZ	Must be zero.
23	<31>	PME	Performance Monitor Enable controls a signal visible to an external hardware performance monitor. This bit is set to identify those processes for which monitoring is desired and to permit their behavior to be observed without interference from other system activity.

Software symbols for these locations consist of the prefix PTX$L_ and the mnemonic. For example, the PCB offset to R3 is PTX$L_R3. Exceptions are longwords 21 and 23, for which the software symbols are:

PTX$L_PØLRASTL	longword 21
PTX$L_P1LRPME	longword 23

To alter its PØBR, P1BR, PØLR, P1LR, ASTLVL or PME, a process must be executing in kernel mode. It must first store the desired new value in the memory image of the PCB then move the value to the appropriate privileged register. This protocol results from the fact that these are read-only fields (for the context switch instructions) in the PCB.

7.2.3 Process Privileged Registers

The ASTLVL and PME fields of the PCB are contained in registers when the process is running. In order to access them, two privileged registers are provided. Figure 7.3 depicts the AST Level Register.

```
 3
 1                                                         3 2   Ø
+----------------------------------------------------------+-----+
|                 ignored; returns Ø                       |AST- |
|                                                          |  LVL|
+----------------------------------------------------------+-----+
```

(read/write)
Figure 7-3 AST Level Register

An MTPR src,#ASTLVL with src<2:Ø> GEQU 5 results in a reserved operand fault. At bootstrap time, the contents of ASTLVL is 4. Figure 7.4 depicts the Performance Monitor Enable (PME) Register.

```
 3
 1                                                              1 Ø
+--------------------------------------------------------------+-+
|                                                              |P|
|                            MBZ                               |M|
|                                                              |E|
+--------------------------------------------------------------+-+
```

(read/write)
Figure 7-4 Performance Monitor Enable Register

At bootstrap time, PME is cleared.

7.3 ASYNCHRONOUS SYSTEM TRAPS (AST)

Asynchronous system traps are a technique for notifying a process of events that are not synchronized with its execution and initiating processing for asynchronous events with the least possible delay. This delay in delivery of the AST may be due to either process non-residence or an access mode mismatch. The efficient handling of AST's in VAX-11 requires some hardware assistance to detect changes in access mode (current access mode in PSL). A process in any of the four execution access modes (kernel, exec, super, and user) may receive AST's; however, an AST for a less privileged access mode must not be permitted to interrupt execution in a more protected access mode. Since outward access mode transitions occur only in the REI instruction, comparison of the current access mode field is made with a privileged register (ASTLVL) containing the most privileged access mode number for which an AST is pending. If the new access mode is greater than or equal to the pending ASTLVL, an IPL 2 interrupt is triggered to cause delivery of the pending AST.

General Software Flow for AST processing:

1. An event associated with an AST causes software enqueuing of an AST control block to the software PCB and the software sets the ASTLVL field in the hardware PCB to the most privileged access mode for which an AST is pending. If the target process is currently executing, the ASTLVL privileged register also has to be set.

2. When an REI instruction detects a transition to an access mode that can be interrupted by a pending AST, an IPL 2 interrupt is triggered to cause delivery of the AST. Note that the REI instruction does not make pending AST checks while returning to a routine executing on the interrupt stack.

3. The (IPL 2) interrupt service routine should compute the correct new value for ASTLVL that prevents additional AST delivery interrupts while in kernel mode and move that value to the PCB and the ASTLVL register before lowering IPL and actually dispatching the AST. This interrupt service routine normally executes on the kernel stack in the context of the process receiving the AST.

4. At the conclusion of processing for an AST, the ASTLVL is recomputed and moved to the PCB and ASTLVL register by software.

7.4 PROCESS STRUCTURE INTERRUPTS

Two of the software interrupt priorities are reserved for process structure software.

They are:

(IPL 2) - AST delivery interrupt.

> This interrupt is triggered by a REI that detects PSL<CUR_MOD> GEQU ASTLVL and indicates that a pending AST may now be delivered for the currently executing process.

(IPL 3) - Process scheduling interrupt.

> This interrupt is only triggered by software to allow the software running at IPL 3 to cause the currently executing process to be blocked and the highest priority executable process to be scheduled.

7.5 PROCESS STRUCTURE INSTRUCTIONS

Process scheduling software must execute on the interrupt stack (PSL<IS> set) in order to have a non-context-switched stack available for use. If the scheduler were running on a process's kernel stack, then any state information it had there would disappear when a new process is selected. Running on the interrupt stack can occur as the result of the interrupt origin of scheduling events, however some synchronous scheduling requests such as a WAIT service may want to cause rescheduling without any interrupt occurrence. For this reason, the Save Process Context (SVPCTX) instruction can be executed while on either the kernel or the interrupt stack and forces a transition to execution on the interrupt stack.

All of the process structure instructions are privileged and require kernel mode.

LDPCTX Load Process Context

Purpose: restore register and memory management context

Format:

opcode

Operation:

```
if PSL<CUR_MOD> NEQU Ø
          then {privileged instruction fault};
{invalidate per-process translation buffer entries};
!PCB is located by physical address in PCBB
if {internal registers for stack pointers} then
          begin
          KSP <- (PCB);
          ESP <- (PCB+4);
          SSP <- (PCB+8);
          USP <- (PCB+12);
          end;
RØ <- (PCB+16);
R1 <- (PCB+2Ø);
R2 <- (PCB+24);
R3 <- (PCB+28);
R4 <- (PCB+32);
R5 <- (PCB+36);
R6 <- (PCB+4Ø);
R7 <- (PCB+44);
R8 <- (PCB+48);
R9 <- (PCB+52);
R1Ø <- (PCB+56);
R11 <- (PCB+6Ø);
AP <- (PCB+64);
FP <- (PCB+68);
tmp1 <- (PCB+8Ø);
if {tmp1<31:3Ø> NEQU 2} OR {tmp1<1:Ø> NEQU Ø} then
                    {reserved operand abort};
PØBR <- tmp1;
if (PCB+84)<31:27> NEQU Ø then {reserved operand abort};
if (PCB+84)<23:22> NEQU Ø then {reserved operand abort};
PØLR <- (PCB+84)<21:Ø>;
if (PCB+84)<26:24> GEQU 5 then {reserved operand abort};
ASTLVL <- (PCB+84)<26:24>;
tmp1 <- (PCB+88);
tmp2 <- tmp1 + 2**23;
if {tmp2<31:3Ø> NEQU 2} OR {tmp2<1:Ø> NEQU Ø} then
                    {reserved operand abort};
P1BR <- tmp1;
if (PCB+92)<3Ø:22> NEQU Ø then {reserved operand abort};
P1LR <- (PCB+92)<21:Ø>;
PME <- (PCB+92)<31>;
if (PCB+92)<3Ø:22> NEQU Ø then {reserved operand abort};
if PSL<IS> EQLU 1 then
```

```
                    begin
                    ISP <- SP;
                    {interrupts off};
                    PSL<IS> <- Ø;
                    SP <- (PCB);                !get KSP
                    {interrupts on};
                    end;
        -(SP) <- (PCB+76);                      !push PSL
        -(SP) <- (PCB+72);                      !push PC
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

reserved operand
privileged instruction

Opcodes:

Ø6	LDPCTX	Load Process Context

Description:

The Process Control Block is specified by the privileged register Process Control Block 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 PC and PSL are moved from the PCB to the stack, suitable for use by a subsequent REI instruction.

Note:

1. Some processors keep a copy of each of the per-process stack pointers 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

Purpose: save register context

Format:

opcode

Operation:

```
if PSL<CUR_MOD> NEQU 0 then
        {privileged instruction fault};
!PCB is located by physical address in PCBB
if {internal registers for stack pointers} then
        begin
        (PCB) <- KSP;
        (PCB+4) <- ESP;
        (PCB+8) <- SSP;
        (PCB+12) <- USP;
        end;
(PCB+16) <- R0;
(PCB+20) <- R1;
(PCB+24) <- R2;
(PCB+28) <- R3;
(PCB+32) <- R4;
(PCB+36) <- R5;
(PCB+40) <- R6;
(PCB+44) <- R7;
(PCB+48) <- R8;
(PCB+52) <- R9;
(PCB+56) <- R10;
(PCB+60) <- R11;
(PCB+64) <- AP;
(PCB+68) <- FP;
(PCB+72) <- (SP)+;                   !pop PC
(PCB+76) <- (SP)+;                   !pop PSL
If PSL<IS> EQLU 0 then
        begin
        PSL<IPL> <- MAXU(1, PSL<IPL>);
        (PCB) <- SP;                 !save KSP
        KSP <- SP;
        {interrupts off};
        PSL<IS> <- 1;
        SP <- ISP;
        {interrupts on};
        end;
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

privileged instruction

Opcodes:

Ø7	SVPCTX	Save Process Context

Description:

The Process Control Block is specified by the privileged register Process Control Block Base. The general registers are saved into the PCB. The PC and PSL currently on the top of the current stack are popped and stored in the PCB. If a SVPCTX instruction is executed when IS is clear, then IS is set, the interrupt stack pointer activated, and IPL is maximized with 1 because of the switch to the interrupt stack.

Notes:

1. The map, ASTLVL, and PME contents of the 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 into 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.

3. Between the SVPCTX instruction that saves state for one process and the LDPCTX that loads the state of another, the internal stack pointers 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.

7.6 USAGE EXAMPLE

The following example illustrates how the process structure instructions can be used to implement process dispatching software. It is assumed that this simple dispatch routine is always entered via an interrupt.

```
;
;                   ENTERED VIA INTERRUPT
;                   IPL=3

RESCHED:            SVPCTX                      ; Save context in PCB
                      .
                      .
                      .
                    <set state to runnable>
                    <and place current PCB>
                    <on proper RUN queue>
                      .
                      .
                      .
                    <Remove head of highest>
                    <priority, non-empty, >
                    <RUN queue.>
                    MTPR @#PHYSPCB, PCBB        ; Set physical PCB address
                                                ;in PCBB
                    LDPCTX                      ; Load context from PCB
                                                ; For new process
                    REI                         ; Place process in execution
```

CHAPTER 8
SYSTEM ARCHITECTURAL IMPLICATIONS

17-June-8Ø -- Rev 5

8.1 INTRODUCTION

Certain portions of the VAX-11 architecture have implications on the system structure of implementations. There are four broad categories of interaction: data sharing and synchronization, restartability, interrupts and errors. Of these, data sharing is most visible to the programmer.

8.2 DATA SHARING AND SYNCHRONIZATION

The memory system must be implemented such that the granularity of access for independent modification is the byte. Note that this does not imply a maximum reference size of one byte but only that independent modifying accesses to adjacent bytes produce the same results regardless of the order of execution. For example, suppose locations Ø and 1 contain the values 5 and 6. Suppose one processor executes INCB Ø and another executes INCB 1. Then regardless of the order of execution, including effectively simultaneous, the final contents must be 6 and 7.

Access to explicitly shared data that may be written must be synchronized by the programmer or hardware designer. Before accessing shared writeable data, the programmer must acquire control of the data structure. Seven instructions (BBSSI, BBCCI, ADAWI, INSQHI, INSQTI, REMQHI, REMQTI) are provided to allow the programmer to control ("interlock") access to a control variable. These interlocked instructions must be implemented in such a way that read, test, modify, and write happen while other processors and I/O devices are locked out of performing interlocked operations on the same control variable. This is termed an interlocked sequence. Only interlocking operations are locked out by the interlock. On the VAX-11/78Ø, the SBI primitive operations are interlock read and interlock write. The interlocked read operation sets the interlock, and the interlocked write releases it.

BBSSI and BBCCI instructions use hardware provided primitive operations to make a read reference, then test, and then make a write reference to a single bit within a single byte in an interlocked sequence. The ADAWI instruction uses a hardware provided primitive operation to make a read and then a write operation to a single aligned word in an interlocked sequence to allow counters to be maintained without other interlocks. The ADAWI instruction takes the hardware lock on the read of the .mw operand (the second operand which is the one being modified).

The INSQUE and REMQUE instructions provide a series of longword reads and writes in an uninterruptible sequence to allow queues to be maintained without other interlocks in a uniprocessor system.

The INSQHI, INSQTI, REMQHI, and REMQTI instructions use an interlock on the queue header to allow queues to be maintained consistently in a multiprocessor system.

In order to provide a functionality upon which some UNIBUS peripheral devices rely, processors must insure that all instructions making byte or word sized modifying references (.mb and .mw) use the DATIP - DATO(B) functions when the operand physical address selects a UNIBUS device. This constraint does not apply to longword, quadword, field, all floating, or string operations if implemented using byte or word modifying references. This constraint also does not apply to instructions precluded from I/O space references (see Appendix A).

In a multiprocessor system, any software clearing PTE<V> or changing the protection code of a page table entry for system space such that it issues a MTPR xxx,#TBIS must arrange for all other processors to issue a similar TBIS. The original processor must wait until all the other processors have completed their TBIS before it allows access to the system page.

8.3 CACHE

A hardware implementation may include a mechanism to reduce access time by making local copies of recently used memory contents. Such a mechanism is termed a cache. A cache must be implemented in such a way that its existence is transparent to software (except for timing and error reporting/control/recovery). In particular, the following must be true:

1. Program writes to memory followed by starting a peripheral output transfer must output the updated value.

2. Completing a peripheral input transfer followed by the program reading of memory must read the input value.

3. A write or modify followed by a HALT on one processor followed by a read or modify on another processor must read the updated value.

4. A write or modify followed by a power failure followed by restoration of power followed by a read or modify must read the updated value provided that the duration of the power failure does not exceed the maximum non-volatile period of the main memory.

5. In multiprocessor systems, access to variables shared between processors must be interlocked by software executing one of the interlocked instructions (BBSSI, BBCCI, ADAWI, INSQHI,INSQTI,REMQHI,REMQTI).

6. Valid accesses to I/O registers must not be cached.

On the VAX-11/780, this is achieved by a cache that writes through to memory and that watches the memory bus for all external writes to memory.

At bootstrap time, the cache must be either empty or valid.

8.4 RESTARTABILITY

The VAX-11 architecture requires that all instructions be restartable after a fault or interrupt that terminated execution before the instruction was completed. Generally, this means that modified registers are restored to the value they had at the start of execution. For some complex or iterative instructions, indicated in Chapter 4, intermediate results are stored in the general registers. In the latter case memory contents may have been altered but the former case requires that no operand be written unless the instruction can be completed. For most instructions with only a single modified or written operand, this implies special processing only when a multibyte operand spans a protection boundary making it necessary to test accessibility of both parts of the operand.

In order that instructions which store intermediate results in the general registers not compromise system integrity, they must insure that any addresses stored or used are virtual addresses, subject to protection checking, and that any state information stored or used cannot result in a non-interruptable or non-terminating sequence.

Instruction operands that are peripheral device registers having access side effects may produce UNPREDICTABLE results due to instruction restarting after faults or interrupts. In order that software may dependably access peripheral device registers, instructions used to access them must not permit a fault or interrupt after the first I/O space access. The instructions and addressing modes that can be used to meet this condition are listed in Appendix A, "INSTRUCTIONS USABLE TO REFERENCE I/O SPACE."

Memory modifications produced as a side effect of instruction execution, e.g. memory access statistics, are specifically excluded from the constraint that memory not be altered until the instruction can be completed.

Instructions that abort are constrained only to insure memory protection (e.g., registers can be changed).

8.5 INTERRUPTS

Underlying the VAX-11 architectural concept of an interrupt is the notion that an interrupt request is a static condition, not a transient event, which can be sampled by a processor at appropriate times. Further, if the need for an interrupt disappears before a processor has honored an interrupt request, the interrupt request can be removed (subject to implementation dependent timing constraints) without consequence.

In order that software be able to operate deterministically it is necessary that any instruction changing the processor priority (IPL) such that a pending interrupt is enabled must allow the interrupt to occur before executing the next instruction that would have been executed had the interrupt not been pending.

Similarly, instructions that generate requests at the software interrupt levels (See Chapter 6) must allow the interrupt to occur, if processor priority permits, before executing the apparently subsequent instruction.

8.6 ERRORS

Processor errors, if not inconsistent with instruction completion, should create high priority interrupt requests. Otherwise, they must terminate instruction execution with an exception (fault, trap or abort), in which case there may also be an associated interrupt request.

Error notification interrupts may be delayed from the apparent completion of the instruction in execution at the time of the error but if enabled, the interrupt must be requested before processor context is switched, priority permitting.

An example of a case where both an interrupt and an exception are associated with the same event occurs when the VAX-11/780 instruction buffer gets a read data substitution (i.e. read memory data error). In this case the interrupt request associated with error will not be taken if the priority of the running program is high, but an abort will occur when an attempt is made to execute the instruction. However, the interrupt is still pending and will be taken when the priority is lowered.

8.7 I/O STRUCTURE

8.7.1 Introduction

The VAX-11 I/O architecture is very similar to the PDP-11 structure, the principal difference being the method by which processor registers (such as the PSL) are accessed (see Chapter 9). Peripheral device control/status and data registers appear at locations in the physical address space, and can therefore be manipulated by normal memory reference instructions. On the VAX-11/780 implementation, this I/O space occupies the upper half of the physical address space and is 2**29 bytes in length. Use of general instructions permits all the virtual address mapping and protection mechanisms described in Chapter 5 to be used when referencing I/O registers. Note: Implementations that include a cache feature must suppress caching for references in the I/O space.

For any member of the VAX-11 series implementing the UNIBUS, there will be one or more areas of the I/O physical address space each 2**18 bytes in length, which "maps through" to the UNIBUS addresses. The collection of these areas is referred to as the UNIBUS space.

8.7.2 Constraints On I/O Registers

The following is a list of both hardware and programming constraints on I/O registers. These items affect both hardware register design and programming considerations.

1. The physical address of an I/O register must be an integral multiple of the register size in bytes, (which must be a power of two); i.e., all registers must be aligned on natural boundaries.

2. References using a length attribute other than the length of the register and/or unaligned references may produce UNPREDICTABLE results. For example a byte reference to a word-length register will not necessarily respond by supplying or modifying the byte addressed.

3. In all peripheral devices, error and status bits that may be asynchronously set by the device must be cleared by software writing a "1" to that bit position and not affected by writing a "0". This is to prevent clearing bits that may be asynchronously set between reading and writing a register.

4. Only byte and word references of a read-modify-write (i.e., ".mb" or ".mw") type in UNIBUS I/O spaces are guaranteed to interlock correctly. References in the I/O space other than in UNIBUS spaces are UNDEFINED with respect to interlocking. This includes the BBSSI and BBCCI instructions.

5. String, quad, octa, F_floating, D_floating, G_floating, H_floating, and field references in the I/O space result in UNDEFINED behavior.

CHAPTER 9
PRIVILEGED REGISTERS

13-May-81 -- Rev 5.2

9.1 PROCESSOR REGISTER SPACE

The processor register space (PRS) provides access to many types of CPU control and status registers such as the memory management base registers, the PSL, and the multiple stack pointers. These registers are explicitly accessible only by the Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions which require kernel mode privileges.

All the internal processor registers are summarized in the tables at the end of this section. Those which need further explanation are described below. Reference to general registers means RØ through R13, the SP, and the PC (See Chapter 2). Registers referenced by the MTPR and MFPR instructions are designated processor registers, and appear in the processor register space.

9.2 PER-PROCESS REGISTERS AND CONTEXT SWITCHING

There are several per-process registers which are loaded from the PCB during a context load operation and, with the exception of the memory mapping registers and AST level, written back to the PCB during a context save operation (see Chapter 7). Some implementations may copy some or all of these registers from the PCB into scratchpad registers and write them back into the PCB during a context save operation. Other implementations may retain the registers in main memory in the PCB.

For this reason, reading or writing any of these registers via the MFPR or MTPR instruction, or through reference to SP, may or may not read or write the register copy in the current PCB, depending on the implementation. Likewise modifying one of these registers in the PCB will not necessarily update the register which appears in the register space or SP.

An implementation may retain some or all per-process internal registers only in the PCB. In this case, MTPR and MFPR for these registers must access the corresponding PCB location. However, implementations that have internal registers in hardware scratchpads are not required to access the corresponding PCB locations for MTPR and MFPR.

9.3 STACK POINTER IMAGES

Reference to SP (the stack pointer) in the general registers will access one of five possible stack pointers; the user, supervisor, executive, kernel, or interrupt stack pointer, depending on the values of the current mode and IS bits in the PSL (see Chapter 6). Additionally, software can access any of the five stack pointers (including the one currently selected by the current mode and IS bits in the PSL) via the MTPR and MFPR instructions (even on processors that implement the KSP, SSP, ESP, or USP only in the PCB) Results are correct even if the stack pointer specified by the current mode and IS bits in the PSL is referenced in the PRS by an MTPR or MFPR instruction. This means that a MFPR/MTPR to the KSP (if IS=Ø) or the ISP (if IS=1) is equivalent to a MOVL from/to the SP.

9.4 MTPR AND MFPR INSTRUCTIONS

MTPR   Move To Processor Register

Format:

```
        opcode  src.rl, procreg.rl
```

Operation:

```
        if PSL <CUR_MOD> NEQ Ø then {reserved
                instruction fault};
        PRS[procreg] <- src;
```

Condition Codes:

```
        N <- src LSS Ø;     !if register is replaced
        Z <- src EQL Ø;
        V <- Ø;             !except TBCHK register (see Chapter 5)
        C <- C;


        N <- N;             !if register is not replaced
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

```
        reserved operand fault
        reserved instruction fault
```

Opcode:

```
  DA    MTPR    Move To Processor Register
```

Description:

Loads the source operand specified by source into the processor register specified by procreg. The procreg operand is a longword which 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
```

Operation:

```
        if PSL <CUR_MOD> NEQ Ø then {reserved
                instruction fault};
        dst <- PRS[procreg];
```

Condition Codes:

```
        N <- dst LSS Ø;         !if destination is replaced
        Z <- dst EQL Ø;
        V <- Ø;
        C <- C;


        N <- N;                 !if destination is not replaced
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

```
        reserved operand fault
        reserved instruction fault
```

Opcode:

```
  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 which 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.

9.5 VAX-11 SERIES REGISTERS

Register Name	Mne-monic	Number	Type	Scope	Init?
Kernel Stack Pointer	KSP	0	R/W	PROC	--
Executive Stack Pointer	ESP	1	R/W	PROC	--
Supervisor Stack Pointer	SSP	2	R/W	PROC	--
User Stack Pointer	USP	3	R/W	PROC	--
Interrupt Stack Pointer	ISP	4	R/W	CPU	--
P0 Base Register	P0BR	8	R/W	PROC	--
P0 Length Register	P0LR	9	R/W	PROC	--
P1 Base Register	P1BR	10	R/W	PROC	--
P1 Length Register	P1LR	11	R/W	PROC	--
System Base Register	SBR	12	R/W	CPU	--
System Limit Register	SLR	13	R/W	CPU	--
Process Control Block Base	PCBB	16	R/W	PROC	--
System Control Block Base	SCBB	17	R/W	CPU	--
Interrupt Priority Level	IPL	18	R/W	CPU	yes
AST Level	ASTLVL	19	R/W	PROC	yes
Software Interrupt Request	SIRR	20	W	CPU	--
Software Interrupt Summary	SISR	21	R/W	CPU	yes
Interval Clock Control	ICCS	24	R/W	CPU	yes
Next Interval Count	NICR	25	W	CPU	--
Interval Count	ICR	26	R	CPU	--
Time of Year (optional)	TODR	27	R/W	CPU	no
Console Receiver C/S	RXCS	32	R/W	CPU	yes
Console Receiver D/B	RXDB	33	R	CPU	--
Console Transmit C/S	TXCS	34	R/W	CPU	yes
Console Transmit D/B	TXDB	35	W	CPU	--
Memory Management Enable	MAPEN	56	R/W	CPU	yes
Trans. Buf. Invalidate All	TBIA	57	W	CPU	--
Trans. Buf. Invalidate Single	TBIS	58	W	CPU	--
Performance Monitor Enable	PMR	61	R/W	PROC	yes
System Identification	SID	62	R	CPU	no
Translation Buffer Check	TBCHK	63	W	CPU	--

9.5.1 System Identification Register (SID)

The SID is a read only constant register that specifies the processor type. The entire SID register is included in the error log and the type field may be used by software to distinguish processor types.

```
 3             2 2
 1             4 3                                        Ø
+--------------+------------------------------------------+
|     TYPE     |              type specific               |
+--------------+------------------------------------------+
                        (read only)
```

System Identification Register

Type   A unique number assigned by engineering to identify a specific processor:

Ø = Reserved to DIGITAL (error)
1 = VAX-11/78Ø
2 = VAX-11/75Ø
3 = VAX-11/73Ø
4 through 127 = Reserved to DIGITAL
128 through 255 = Reserved to CSS and customers

type specific   format and content is a function of the value in type. It is intended to include such information as serial number and revision level.

For the VAX-11/78Ø, the type specific format is:

```
 2              1 1   1 1
 3              5 4   2 1                   Ø
+---------------+-----+---------------------+
|  ECO level    |plant|    serial number    |
+---------------+-----+---------------------+
```

For the VAX-11/75Ø, the type specific format is:

```
 2            1 1
 3            6 5            8 7            Ø
+-------------+--------------+--------------+
|             | microcode rev | hardware rev |
+-------------+--------------+--------------+
```

9.5.2 Console Terminal Registers

The console terminal is accessed through four internal registers. Two are associated with receiving from the terminal and two with writing to the terminal. In each direction there is a control/status register and a data buffer register.

```
 3
 1                                               8 7 6 5           Ø
+-----------------------------------------------+-+-+-----------+
|                                               |D|I|           |
|                     MBZ                       |O|E|    MBZ    |
|                                               |N| |           |
+-----------------------------------------------+-+-+-----------+
                                                 R R
                                                 O W
```

Console Receive Control/Status (RXCS)

```
 3                             1 1 1   1 1
 1                             6 5 4   2 1     8 7            Ø
+-----------------------------+-+-----+-------+---------------+
|                             |E|     |       |               |
|              Ø              |R|  Ø  |  ID   |     DATA      |
|                             |R|     |       |               |
+-----------------------------+-+-----+-------+---------------+
```

(read only)
Console Receive Data Buffer (RXDB)

At bootstrap time, RXCS is initialized to Ø. Whenever a datum is received, the read only bit DONe is set by the console. If IE (interrupt enable) is set by the software then an interrupt is generated at IPL 2Ø. Similarly, if DONe is already set and the software sets IE, an interrupt is generated (i.e., an interrupt is generated whenever the function {IE AND DON} changes from Ø to 1). If the received data contained an error such as overrun or loss of connection then ERR is set in RXDB. The received data appears in DATA. When a MFPR #RXDB,dst is executed, DONe is cleared as is any interrupt request. If ID is Ø then the data is from the console terminal. If ID is non-zero then the entire register is implementation dependent.

```
 3
 1
                                               8 7 6 5           Ø
+----------------------------------------------+-+-+-------------+
|                                              |R|I|             |
|                     MBZ                      |D|E|    MBZ      |
|                                              |Y| |             |
+----------------------------------------------+-+-+-------------+
                                                R R
                                                O W
```

Console Transmit Control/Status (TXCS)

```
 3
 1                                      1 1
                                        2 1     8 7              Ø
+---------------------------------------+-------+----------------+
|                  MBZ                  |  ID   |      DATA      |
+---------------------------------------+-------+----------------+
```

(write only)
Console Transmit Data Buffer (TXDB)

At bootstrap time, TXCS is initialized with just the RDY bit set (ready). Whenever the console transmitter is not busy, it sets the read only bit RDY. If IE (interrupt enable) is set by the software then an interrupt is generated at IPL 2Ø. Similarly, if RDY is already set and the software sets IE, an interrupt is generated (i.e., an interrupt is generated whenever the function {IE AND RDY} changes from Ø to 1). The software can send a datum by writing it to DATA. When a MTPR src,#TXDB is executed, RDY is cleared as is any interrupt request. If ID is written Ø then the datum is sent to the console terminal. If ID is non-zero then the entire register is implementation dependent.

On the VAX-11/78Ø if ID is one then the datum is sent to the floppy disk.

9.5.2.1 VAX-11/78Ø console register implementation - RXDB

```
 3         2 2         1 1         1 1
 1         4 3         6 5         2 1   8 7            Ø
+----------+-----------+-----------+-----+-------------+
|          |           |           |     |             |
|   MBZ    |    MBZ    |  Used by  |     |             |
|          |           |   DL-11   |     |             |
+----------+-----------+-----------+-----+-------------+
                                      |          |
                                      |          |
                                   Select      Data
                                   Field       Field
```

TXDB

```
 3         2 2         1 1         1 1    8 7          0
 1         4 3         6 5         2 1                  
+----------+-----------+-----------+-----+------------+
|          |           |           |     |            |
|   MBZ    |    MBZ    |    MBZ    |     |            |
|          |           |           |     |            |
+----------+-----------+-----------+-----+------------+
                                      |         |
                                      |         |
                                   Select     Data
                                   Field      Field
```

Select Field Values (in Hex)

Select Code	Device	Data Field Values
0	Operator's Terminal	0 thru 7F - ASCII Data
1	Drive 0 (Data)	0 thru FF - Binary Data
2	Function Complete	(Status)
9	Drive 0 (Command)	0 = Read Sector 1 = Write Sector 2 = Read Status 3 = Write Deleted Data Sector 4 = Cancel Function 5 = Protocol Error
F	Misc. Communication	1 = Software Done 2 = Boot CPU 3 = Clear Warm-start flag 4 = Clear Cold-start flag

Code 5 (Protocol Error), is sent by the console when one of the following occurs:

1. Another load device command (except for Cancel Function) is issued by the OS before a previous command is completed.

2. The console gets a 'Drive 0 (DATA)' when expecting a command.

9.5.2.1.1 Status Byte Definition - The Status Byte is used by VMS to determine the success or failure of a Read or Write operation. The Status Byte is sent to the OS at the completion of a Read, Write, or Read Status operation. The Select code is always 'Function Complete' (code 2). The Status Bit assignments are as follows:

RXDB

```
 3           2 2           1 1          1 1
 1           4 3           6 5          2 1  8 7 6    2 1 Ø
+------------+------------+------------+-----+-+-+---+-+-+-+
|            |            |            |     | | |   | | | |
|   MBZ      |    MBZ     |    MBZ     |     | | |   | | | |
|            |            |            |     | | |   | | | |
+------------+------------+------------+-----+-+-+---+-+-+-+
                                        |     | |     | | |
                           CODE '2'     |     | |     | | | CRC ERR
                           ------------|     | |     | | |--------
                                              | |     | |
                                              | |     | | PARITY ERROR
                                              | |     | |-------------
                                              | |     | INI DONE
                                              | |     |---------
                                              | | DELETED DATA
                                              | |-------------
                                              | ERROR
                                              |------
```

The Status Bit assignments are identical to those supplied by the Floppy controller, excepting Bit 7. Bit 7 corresponds to Bit 15 of the Floppy's 'RXCS' Register.

9.5.3 Clock Registers

The clocks consist of a time of year clock and an interval clock. The time of year clock is used to measure the duration of power failures and is required for unattended restart after a power failure. The interval clock is used for accounting, for time dependent events, and to maintain the software date and time.

9.5.3.1 Time-of-Year Clock -

The time-of-year clock consists of one longword register. The register forms an unsigned 32-bit binary counter that is driven by a precision clock source with at least .ØØ25% accuracy (approximately 65 seconds per month). The least significant bit of the counter represents a resolution of 1Ø milliseconds. Thus, the counter cycles to Ø after approximately 497 days.

The counter has an optional battery back-up power supply sufficient for at least 100 hours of operation, and the clock does not gain or lose any ticks during transition to or from stand-by power. The battery is recharged automatically. If the battery has failed, so that time is not accurate, then the register is cleared upon power up. One of two things then happens:

1. The register starts counting from 0. Thus, if software initializes this clock to a value corresponding to a large time (e.g., a month), it can check for loss of time after a power restore by checking the clock value. This is the VAX-11/780 implementation.

2. The register stays at 0 until the software writes a non-zero value into it. It counts only when it contains a non-zero value. This is the VAX-11/750 implementation.

```
 3
 1                                                              0
+-----------------------------------------------------------------+
|                  time of year since setting                     |
+-----------------------------------------------------------------+
                             (read/write)

                          Time of Year (TODR)
```

9.5.3.2 Interval Clock - The interval clock provides an interrupt at IPL 24 at programmed intervals. The counter is incremented at 1 microsecond intervals, with at least .01% accuracy (8.64 seconds per day). The clock interface consists of three registers in the privileged register space:

```
 3
 1                                                              0
+----------------------------------------------------------------+
|                        interval count                          |
+----------------------------------------------------------------+
                          (read only)
                  interval count register (ICR)

 3
 1                                                              0
+----------------------------------------------------------------+
|                      next interval count                       |
+----------------------------------------------------------------+
                         (write only)
                      next interval (NICR)

 3 3
 1 0                                         8 7 6 5 4 3   1 0
+-+-+----------------------------------------+-+-+-+-+-----+-+-+
|E|                                          |I|I|S|X|     |R|
|R|                    MBZ                   |N|E|G|F| MBZ |U|
|R|                                          |T| |L|R|     |N|
+-+-+----------------------------------------+-+-+-+-+-----+-+-+
 W                                            W R W W       R
 C                                            C W O O       W

               Interval Clock Control/Status (ICCS)
```

1. Interval Count - The interval register is a read only register incremented once every microsecond. It is automatically loaded from NICR upon a carry out from bit 31 (overflow) which also interrupts at IPL 24 if the interrupt is enabled.

2. Next Interval Count - The reload register is a write only register that holds the value to be loaded into ICR when it overflows. The value is retained when ICR is loaded. NICR is capable of being loaded regardless of the current values of ICR and ICCS.

3. Interval Clock Control Status (ICCS) - The ICCS register contains control and status information for the interval clock.

RUN <0>	When set, ICR increments each microsecond. When clear, ICR does not increment automatically. At bootstrap time, run is cleared.

XFR	<4>	A write only bit. Each time this bit is set, NICR is transferred to ICR.
SGL	<5>	A write only bit. If RUN is clear, each time this bit is set, ICR is incremented by one.
IE	<6>	When set, an interrupt request at IPL 24 is generated every time ICR overflows (INT is set). When clear, no interrupt is requested. Similarly, if INT is already set and the software sets IE, an interrupt is generated (i.e., an interrupt is generated whenever the function {IE AND INT} changes from Ø to 1).
INT	<7>	Set by hardware every time ICR overflows. If IE is set then an interrupt is also generated. An attempt to set this bit via MTPR clears INT, thereby reenabling the clock tick interrupt (if IE is set).
ERR	<31>	Whenever ICR overflows, if INT is already set, then ERR is set. Thus, ERR indicates a missed clock tick. An attempt to set this bit via MTPR clears ERR.

Thus, to setup the interval clock, load the negative of the desired interval into NICR. Then a MTPR #^X51,#ICCS will enable interrupts, reload ICR with the NICR interval and set run. Every "interval count" microseconds will cause INT to be set and an interrupt to be requested. The interrupt routine should execute a MTPR #^XC1,#ICCS to clear the interrupt. If INT has not been cleared (i.e., the interrupt has not been handled) by the time of the next ICR overflow, the ERR bit will be set.

At bootstrap time, bits <6> and <Ø> of ICCS are cleared. The rest of ICCS and the contents of NICR and ICR are UNPREDICTABLE.

9.6 VAX-11/780 SPECIFIC REGISTERS

Register Name	Mne-monic	Number	Type	Scope	Init?
Accelerator Control/Status	ACCS	40	R/W	CPU	yes
Accelerator Maintenance	ACCR	41	R/W	CPU	no
WCS Address	WCSA	44	R/W	CPU	no
WCS Data	WCSD	45	R/W	CPU	yes
SBI Fault/Status	SBIFS	48	R/W	CPU	yes
SBI Silo	SBIS	49	R	CPU	no
SBI Silo Comparator	SBISC	50	R/W	CPU	yes
SBI Maintenance	SBIMT	51	R/W	CPU	yes
SBI Error Register	SBIER	52	R/W	CPU	yes
SBI Timeout Address	SBITA	53	R	CPU	--
SBI Quadword Clear	SBIQC	54	W	CPU	--
Micro Program Breakpoint	MBRK	60	R/W	CPU	no

9.6.1 VAX-11/780 Accelerator

The VAX-11/780 processor has an optional accelerator for a subset of the instructions. Two internal registers control the accelerator, ACCS and ACCR.

ACCS is the accelerator control and status register. It indicates whether an accelerator exists, controls whether it is enabled, identifies its type and reports errors and status. At bootstrap time, the type and enable are set; the errors are cleared.

```
 3 3 2 2 2 2                     1 1 1
 1 0 9 8 7 6                     6 5 4           8 7              0
+-+-+-+-+-+-+---------------------+-+-------------+----------------+
|E|M|U|O|R|                       |E|             |                |
|R|B|N|V|S|         MBZ           |N|     MBZ     |      TYPE      |
|R|Z|F|F|V|                       |B|             |                |
+-+-+-+-+-+-+---------------------+-+-------------+----------------+
 R   R R R                         R                      RO
 O   O O O                         W
```

Accelerator Control/Status (ACCS)

TYPE <7:0> Read only field specifying the accelerator type as follows:

0 = No Accelerator
1 = Floating point accelerator

Numbers in the range 2 through 127 are reserved to DIGITAL. Numbers in the range 128 through 255 are reserved to CSS/customers.

ENB <15> Read/write field specifying whether the accelerator is enabled. At bootstrap time, this is set if the accelerator is installed and functioning. An attempt to set this if no accelerator is installed is ignored.

RSV <27> Read only bit specifying that the last operation had a reserved operand.

OVF <28> Read only bit specifying that the last operation had an overflow.

UNF <29> Read only bit specifying that the last operation had an underflow.

ERR <31> Read only bit specifying that at least one of bits RSV, OVF, and UNF is set. Note that bits <31:27> are normally cleared by the main processor microcode before starting the next macro instruction.

ACCR is the accelerator maintenance register. It controls the accelerator's microprogram counter. At bootstrap time its contents are UNPREDICTABLE.

```
 3 3           2 2             1 1 1 1
 1 0           4 3             6 5 4 3      9 8              0
+-+-------------+---------------+-+-+---------+-----------------+
|E|             |               |E|M|         |                 |
|T|     MBZ     | TRAP ADDRESS  |M|P|   MBZ   |    MICRO PC     |
|L|             |               |L|M|         |                 |
+-+-------------+---------------+-+-+---------+-----------------+
 W                    RW         W R                 RW
 O                               O O
```

Accelerator Maintenance Register (ACCR)

PC <0:8> NEXT MICRO PC on read. This contains the next micro address to be executed.

MATCH MICRO PC on write. If EML is also set, then this updates the micro PC match register.

MPM <14> MICRO PC MATCH. A read only bit that is set whenever the accelerator's micro PC matches the micro PC match register. This is useful primarily as a scope sync signal.

EML <15> ENABLE MICRO PC MATCH LOAD. A write only bit that when set causes <8:0> to be loaded into the accelerator's micro PC match register.

TRAP <16:23> TRAP ADDRESS. A read/write field used by the main processor to force the accelerator to a specified micro location.

ETL <31> ENABLE TRAP ADDRESS LOAD. A write only bit that when set causes <23:16> to be loaded into the accelerator's trap address register. Subsequently, the main processor's micro code can force the accelerator to trap to this location by asserting an internal signal.

9.6.2 VAX-11/780 Micro Control Store

The VAX-11/780 processor has three registers for control/status of its microcode. Two are used for writing into any writable control store (WCS) and one is used to control micro breakpoints.

```
 3                             1 1 1 1 1
 1                             6 5 4 3 2                    0
+-----------------------------+-+---+-------------------------+
|                             |P|   |                         |
|          MBZ                |I|CTR|       WCS ADDR          |
|                             |N|   |                         |
+-----------------------------+-+---+-------------------------+
                               R RW            RW
                               W
```

Writable Control Store Address (WCSA)

```
 3
 1                                                           0
+-------------------------------------------------------------+
|                         WCS Data                            |
+-------------------------------------------------------------+
```

(on Write)

```
 3
 1                                             8 7           0
+---------------------------------------------+---------------+
|                    0                        |    PRESENT    |
+---------------------------------------------+---------------+
```

(on read)
Writable Control Store Data (WCSD)

Reading WCSD indicates which control store addresses are writable. If WCSD<n> is set, then addresses n*1024 through n*1024+1023 are writable (i.e., that WCSA<12:10> EQLU n corresponds to writable control store). n=4 corresponds to WCS that is reserved to DIGITAL for diagnostics and engineering change orders. Other fields correspond to blocks of control that can be used to implement customer or CSS specific microcode. Each word of control store contains 96 bits plus 3 parity bits. To write one or more words, initialize WCS ADDR to the address and CTR to 0. Then each MTPR to WCSD will write the next 32 bits and automatically increment CTR. When CTR would become 3, it is automatically cleared and

WCS ADDR is incremented. If PIN is set, then any writes to WCSD are done with inverted parity. An attempt to execute a microword with bad parity results in a machine check. At bootstrap time, the contents of WCSA are UNPREDICTABLE.

```
 3                                   1 1                        0
 1                                   3 2                        
+-----------------------------------+------------------------+
|                MBZ                |        MICRO PC        |
+-----------------------------------+------------------------+
                        (read/write)
```

Micro Program Breakpoint Address (MBRK)

Whenever the microprogram PC matches the contents of MBRK, an external signal is asserted. If the console has enabled stop on microbreak, then the processor clock is stopped when this signal is asserted. If the console has not enabled microbreak, then this signal is available as a diagnostic scope point. Many diagnostics use the NOP instruction to trigger this method of giving a scope point. At bootstrap time, the contents of MBRK are UNPREDICTABLE.

9.6.3 SBI FAULT/STATUS REGISTER (SBIFS)

```
 3 3 2 2 2 2 2 2        2 1 1 1 1 1                               0
 1 0 9 8 7 6 5 4        0 9 8 7 6 5
+-+-+-+-+-+-+-+-+---------+-+-+-+-+-------------------------------+
|P|M|U|M|M|X|N|         |L|I|S|S|                               |
|T|B|N|B|L|M|S|   MBZ   |T|N|I|I|              MBZ              |
|Y|Z|X|Z|T|T|T|         |H|T|G|L|                               |
+-+-+-+-+-+-+-+-+---------+-+-+-+-+-------------------------------+
 R   R   R R             W   R W
 O   O   O O             C   O C
```

15:0	MBZ		
16	SIL	SILO FLT LOCK	Fault Silo Lock (set if Silo Locked due to Fault Signal)
17	SIG	SIG FLT	Fault Signal
18	INT	INT FLT EN	Fault Interrupt Enable
19	LTH	LTH FLT	Fault Latch
20:24	MBZ		
25	NST	NST ERR	Nested Error
26	XMT	XMT FLT	Transmitter during Fault cycle
27	MLT	MLT XMT	Multiple Transmitter Fault Flag
29	UNX	UNX RD	Unexpected read Data Fault Flag
31	PTY	PTY FLT	SBI Parity Fault Flag

9.6.4 SBI SILO DATA REGISTER (SBIS)

The SBI Silo is a history of the state of the indicated SBI signals for the past 16 SBI cycles. The silo is updated every cycle until FAULT is asserted on the SBI or an SBI Silo Comparator match occurs. Each entry in the silo has the following format:

```
 3 3 2         2 2   2 2    1 1 1 1
 1 0 9         5 4   2 1    8 7 6 5                              0
+-+-+----------+-----+-------+---+--------------------------------+
|A|I|          |     |       |   |                                |
|F|N|    ID    | TAG |  SBI  |CNF|          SBI TR<15:0>          |
|T|T|          |     |       |   |                                |
+-+-+----------+-----+-------+---+--------------------------------+
```

READ ONLY

0:15	SBI TR		SBI Transmit/Receive Lines
17:16	CNF	SBI CNF1-0	SBI Confirmation Lines
21:18	SBI	SBI M3-M0 OR B31-B28	SBI bits 21-18 are written with SBI B31-B28 when SBI TAG FIELD specifies command address TAG. Otherwise, M3-M0 are written in this field.
24:22	TAG	SBI TAG	
29:25	ID	SBI IDI	
30	INT	INTLK	SBI Interlock
31	AFT	AFT FLT	First Entry after FAULT cleared

9.6.5 SBI SILO COMPARATOR REGISTER (SBISC)

The Silo Comparator allows the SBI to become locked when pre specified conditions are detected. Conditional and unconditional lock modes are provided by the Silo Comparator.

```
 3 3 2 2 2 2       2 2   2 1     1 1
 1 0 9 8 7 6       3 2   0 9     6 5                              0
+-+-+-+---+-------+-----+-------+-------------------------------+
|C|I|L| L |COMPARE| COMP| COUNT |                               |
|M|N|C| O | CMD OR|     |       |              MBZ              |
|P|T|K| C |  MASK | TAG | FIELD |                               |
+-+-+-+---+-------+-----+-------+-------------------------------+
 *
                  *CLEARED ON ANY WRITE TO SBISC
```

15:0	MBZ		
19:16	COUNT FIELD		
22:20	COMPARE TAG		
26:23	COMPARE CMD or MASK		Command or Mask
28:27	LOC	LOCK COND CODES	Conditional Lock Codes
29	LCK	LCK UNCOND	Lock Unconditional
30	INT	INT EN	Silo Lock Interrupt Enable
31	CMP	CMP SILO LOCK	Compare Silo Lock

9.6.6 SBI MAINTENANCE REGISTER (SBIMT)

The SBI Maintenance register allows error conditions to be forced for diagnostic purposes.

```
 3 3 2 2 2        2 2 2 2      1 1 1 1 1 1 1 1
 1 Ø 9 8 7        3 2 1 Ø      7 6 5 4 3 2 1 Ø 9 8 7                Ø
+-+-+-+-+----------+-+-+--------+-+-+-+-+-+-+-+-+-+-+----------------+
|P|W|U|M|          |I|E|        |F|F|F|F|D|P|G|G|T|                |
|Ø|R|N|L| MAINT ID|N|N|  REV   |G|G|R|R|S|1|1|Ø|I|      MBZ       |
| |T|X|T|          |V|I|        |Ø|1|Ø|1|B| | | |M|                |
+-+-+-+-+----------+-+-+--------+-+-+-+-+-+-+-+-+-+-+----------------+
 R R R R     R      R R   R              R R R R R
 O O O O     O      O O   O              O O O O O
```

Ø:7	MBZ		
8	TIM	TIME F OUT	Force Timeout on Read
9	GØ	GØ MAT	Group Ø Match
1Ø	G1	G1 MAT	Group 1 Match
11	P1	REV SBI P1	Force P1 reversal on SBI
12	DSB	DSBL SBI CYC	Disable SBI Cycles
13	FR1	F G1 REP	Force Cache Replacement Group 1
14	FRØ	F GØ REP	Force Cache Replacement Group Ø
15	FG1	F G1 MISS	Force Cache Miss Group 1
16	FGØ	MISS F GØ	Force Cache Miss Group Ø
2Ø:17	REV	REV CACHE PAR FIELD	Reverse Cache Parity Field
21	ENI	EN SBI INV	Enable SBI Invalidate
22	INV	INV F SBI	Force SBI Write Invalidate to Cache
27:23	MAINT ID		Maintenance ID - to force faults and as SILO Comparator
28	MLT	MLT F XMIT	Force Multiple Transmitter Fault
29	UNX	UNEX F RD	Force Unexpected Read Data Fault
3Ø	WRT	WRT F SEQ	Force Write Sequence Fault
31	PØ	PØ REV SBI	Force PØ Reversal on SBI

9.6.7 SBI ERROR REGISTER (SBIER)

```
 3                             1 1 1 1 1 1 1
 1                             6 5 4 3 2 1 Ø 9 8 7 6 5 4 3 2 1 Ø
+------------------------------+-+-+-+-+---+-+-+-+-+---+-+-+-+-+
|                              |C|C|R|C|CP |M|C|I|I|IB |I|M|I|M|
|             MBZ              |I|R|D|P|TIM|B|E|R|B|TIM|E|L|N|B|
|                              |E|D|S| |OUT|Z|C| | |OUT|C|T|B|Z|
+------------------------------+-+-+-+-+---+-+-+-+-+---+-+-+-+-+
                                 W W W R R   R W W   R   R R R
                                 C C C O O   O C C   O   O O O
```

Ø	MBZ		
1	INB	INT NOT BSY SBT	SBI Interface Not Busy
2	MLT	MLT CP ERR	Multiple CP Error
3	IEC	IB SBI CNF ERR	Error Confirmation
5:4	IB TIM OUT	IB TIME OUT STATUS	
6	IB	IB TIME OUT	
7	IR	IB RDS	
8	CEC	CP SBI CNF ERR	Error Confirmation
9	MBZ		
11:1Ø	CP TIM OUT	CP TIME OUT STATUS	
12	CP	CP TIME OUT	
13	RDS		Read Data Substitute - set whenever RDS is returned to CPU.
14	CRD		CRD (Corrected Read Data) is set whenever CRD is returned to CPU.
15	CIE	CRD INT EN RDS	CRD/RDS Interrupt Enable
31:16	MBZ		

9.6.8 SBI TIMEOUT ADDRESS (SBITA)

This register is a holding register for the Physical Address sent on the SBI. When a timeout occurs on the SBI, this register will latch up with the physical address of the timeout. It is reset by clearing bit 12 of the SBI error register.

```
 3 3 2 2 2
 1 0 9 8 7                                                  0
+-+-+-+-+-----------------------------------------------------+
|M|M|P| |                                                     |
|1|0|C|0|             PHYSICAL ADDRESS <29:2>                 |
+---+-+-+-----------------------------------------------------+

                       READ ONLY
```

27:0	PHYSICAL ADDRESS <29:2>		
28	0		
29	PC	NO PROT CHK	Protection checked reference.
30	M0		Mode 0 reference
31	M1		Mode 1 reference

9.6.9 SBI QUAD CLEAR (SBIQC)

```
 3 3 2
 1 0 9                                                    3 2   0
+---+-----------------------------------------------------+-----+
|MBZ|            PHYSICAL QUADWORD ADDRESS                 | MBZ |
+---+-----------------------------------------------------+-----+

                       WRITE ONLY
```

9.7 VAX-11/750 SPECIFIC REGISTERS

Register Name	Mne-monic	Number	Type	Scope	Init?
CMI Error Register	CMIERR	23	R	CPU	yes
Console Storage Receiver Status	CSRS	28	R/W	CPU	yes
Console Storage Receiver Data	CSRD	29	R	CPU	--
Console Storage Transmit Status	CSTS	3Ø	R/W	CPU	yes
Console Storage Transmit Data	CSTD	31	W	CPU	--
Translation Buffer Disable	TBDR	36	R/W	CPU	--
Cache Disable	CADR	37	R/W	CPU	--
Machine Check Error Summary	MCESR	38	R/W	CPU	--
Cache Error	CAER	39	R/W	CPU	--
Accelerator Control/Status	ACCS	4Ø	R/W	CPU	--
Initialize UNIBUS	IORESET	55	W	CPU	--
Translation Buffer Data	TBDATA	59	R/W	CPU	--

9.7.1 CMI Error Register

```
 3                    2 2 1     1 1   1 1 1
 1                    1 Ø 9     6 5   3 2 1      8 7   5 4 3      Ø
+--------------------+-+--------+-----+-+--------+-----+-+--------+
|         Ø          | | SMR    |  Ø  | | TBGPR  |  Ø  | |  BER   |
+--------------------+-+--------+-----+-+--------+-----+-+--------+
```

Ø:3	BER	Bus Error
Ø		Corrected Data Error
1		Lost Error
2		Uncorrectable Data Error
3		Non-existent memory
4	TBHIT	TB hit on last reference
11:8	TBGPR	TB Group Error
8		TB Group Ø Data error
9		TB Group 1 Data Error
1Ø		TB Group Ø Tag Error
11		TB Group 1 Tag error
12	RLTO	Read Lock Timeout
18:16	SMR	Saved Mode Register
17:16		Processor access mode for last reference
18		Virtual=Ø, Physical=1
2Ø	CMIDIS	Disable CMI references

9.7.2 Console Storage Device Registers

The VAX-11/75Ø accesses the console storage device through four internal registers that are distinct from those used to access the console terminal. The architecture of these registers is similar to that of the console terminal registers.

```
 3
 1                                                    8 7 6 5           Ø
+----------------------------------------------------+-+-+-------------+
|                                                    |D|I|             |
|                       MBZ                          |O|E|     MBZ     |
|                                                    |N| |             |
+----------------------------------------------------+-+-+-------------+
                                                      R R
                                                      O W
```

Console Storage Receive Status (CSRS)

```
 3
 1                                                    8 7               Ø
+----------------------------------------------------+-----------------+
|                       Ø                            |       DATA      |
+----------------------------------------------------+-----------------+
```

(read only)
Console Storage Receive Data Buffer (CSRD)

```
 3
 1                                                    8 7 6 5         1 Ø
+----------------------------------------------------+-+-+-----------+-+
|                                                    |R|I|           |B|
|                       MBZ                          |D|E|    MBZ    |R|
|                                                    |Y| |           |K|
+----------------------------------------------------+-+-+-----------+-+
                                                      R R             W
                                                      O W             O
```

Console Storage Transmit Status (CSTS)

```
 3
 1                                                    8 7               Ø
+----------------------------------------------------+-----------------+
|                       Ø                            |       DATA      |
+----------------------------------------------------+-----------------+
```

(write only)
Console Storage Transmit Data Buffer (CSTD)

9.7.3 Translation Buffer Group Disable Register (TBDR)

```
 3
 1                                                          4 3 2 1 0
+----------------------------------------------------------+-+-+-+-+-+
|                             0                            | | | | | |
+----------------------------------------------------------+-+-+-+-+-+
```

```
0               Force Miss Group 0
1               Force Miss Group 1
2               if {<3> EQL 1} then this bit selects group to be replaced
3               0 = Random replacement, 1 = Force replacement
```

9.7.4 Cache Disable Register (CADR)

```
 3
 1                                                                1 0
+----------------------------------------------------------------+-+-+
|                            MBZ                                 | | |
+----------------------------------------------------------------+-+-+
```

```
0               Disable cache
```

9.7.5 Machine Check Error Summary Register (MCESR)

```
 3
 1                                                          4 3 2 1 0
+----------------------------------------------------------+-+-+-+-+-+
|                             0                            | | |0| | |
+----------------------------------------------------------+-+-+-+-+-+
```

```
0               Reference was through prefetch logic
2               TB parity error
3               Bus error
```

9.7.6 Cache Error Register (CAER)

```
 3
 1
                                                        4 3 2 1 Ø
+--------------------------------------------------------+-+-+-+-+
|                        Ø                               | | | | |
+--------------------------------------------------------+-+-+-+-+
```

Ø	Cache hit
1	Lost error
2	Cache data parity error
3	Cache Tag parity error

9.7.7 Accelerator Control/Status Register

The accelerator control and status register on the VAX-11/75Ø is a subset of the ACCS on the VAX-11/78Ø.

```
 3                          1 1 1
 1                          6 5 4           8 7             Ø
+----------------------------+-+-------------+---------------+
|                            |E|             |               |
|            MBZ             |N|     MBZ     |     TYPE      |
|                            |B|             |               |
+----------------------------+-+-------------+---------------+
                              W                    RO
                              O
```

<7:Ø>	TYPE	Ø = no accelerator or disabled 1 = Floating Point Accelerator (FPA) Numbers in the range 2-127 are reserved to DIGITAL. Numbers in the range 128-255 are reserved to CSS/customers.
<15>	ENB	Enable FPA.

ACCS<15> always reads as Ø. To determine if an FPA is present, write a 1 to ACCS<15> and then read ACCS<Ø>. If there is no FPA, ACCS<Ø> will read as Ø.

9.7.8 Initialize UNIBUS (IORESET)

```
 3
 1
                                                            1 Ø
+----------------------------------------------------------+-+
|                      MBZ                                 | |
+----------------------------------------------------------+-+
```

<Ø>   Initialize Unibus

9.7.9 Translation Buffer Data Register (TBDATA)

This internal processor register is used to read and write locations in the TB. On a MFPR to this register, the page table entry for the virtual address in PØBR is read from the TB into the destination. On a MTPR, the source operand is written into the TB as the page table entry for the virtual address in PØBR. The results of an MTPR/MFPR on the register are UNPREDICTABLE if memory management is enabled.

CHAPTER 10
PDP-11 COMPATIBILITY MODE

23-March-81 -- Rev 5.2

10.1 INTRODUCTION

VAX compatibility mode hardware, in conjunction with a compatibility mode software executive (which runs in VAX mode), can emulate the environment provided to user programs on a PDP-11. This environment excludes the following features of normal PDP-11 operation:

1. Privileged instructions such as HALT and RESET.
2. Special instructions such as traps and WAIT.
3. Access to internal processor registers (e.g., PSW and console switch register).
4. Direct access to trap and interrupt vectors.
5. Direct access to I/O devices.
6. Interrupt servicing.
7. Stack overflow protection.
8. Alternate general register sets.
9. Any processor mode other than user (i.e., Kernel and Supervisor modes are not supported) and separate I and D spaces.
10. Floating point instructions.

This specification is based on the behavior of all PDP-11 implementations. Compatibility mode behavior is defined as UNPREDICTABLE where there is a difference between any two PDP-11 implementations.

10.2 COMPATIBILITY MODE USER ENVIRONMENT

10.2.1 General Registers And Addressing Modes

All of the PDP-11 general registers and addressing modes are provided in compatibility mode. Side effects caused by a destination address calculation have no effect on source values (except in JSR), and auto-increment modes in JMP and JSR do not affect the new PC. However, side effects caused by a source address calculation might affect the value of a register used for destination address calculation. All PDP-11 addresses are 16 bits wide. In compatibility mode, a 16-bit PDP-11 address is zero-extended to 32 bits.

10.2.1.1 Register Mode -

The addressing format for register mode is:

```
 5   3 2   0
+-----+-----+
|  0  | Rn  |
+-----+-----+
```

In register mode addressing, the operand is the contents of register n:

```
operand = Rn
```

Byte operations, except for MOVB to a register, access the low order byte, i.e. bits <7:0>. The low byte is sign-extended if a register is used as the destination of a MOVB instruction. If the PC is used as the destination of a byte instruction, the result is UNPREDICTABLE.

The assembler notation for register mode is Rn.

10.2.1.2 Register Deferred Mode -

The addressing format for register deferred mode is:

```
 5   3 2   0
+-----+-----+
|  1  | Rn  |
+-----+-----+
```

In register deferred mode addressing, the address of the operand is the contents of register n:

```
OA = Rn

operand = (OA)
```

The assembler notation for register deferred mode is (Rn) or @Rn.

1Ø.2.1.3 Autoincrement Mode -

The addressing format for autoincrement mode is:

```
 5   3 2   Ø
+-----+-----+
|  2  | Rn  |
+-----+-----+
```

If Rn denotes PC, immediate data follows the instruction, and the mode is termed immediate mode.

In autoincrement mode addressing, the address of the operand is the contents of register n. After the operand address is determined, the size of the operand in bytes (1 for byte; 2 for word) is added to the contents of register n (except in the case of SP and PC), and the register is replaced by the result. If Rn denotes SP or PC, the register is incremented by 2 and the register is replaced by the result.

```
OA = Rn

if n LEQ 5 then Rn <- Rn + size else Rn <- Rn + 2

operand = (OA)
```

The assembler notation for autoincrement mode is (Rn)+. For immediate mode the notation is #constant where constant is the immediate data which follows the instruction.

1Ø.2.1.4 Autoincrement Deferred Mode -

The addressing format for autoincrement deferred mode is:

```
 5   3 2   Ø
+-----+-----+
|  3  | Rn  |
+-----+-----+
```

If Rn denotes PC, a 16-bit address follows the instruction, and the mode is termed absolute mode.

In autoincrement deferred mode addressing, the address of the operand is the contents of a word whose address is the contents of register n. After the operand address is determined, 2 is added to the contents of register n, and the register is replaced by the result.

```
OA = (Rn)

Rn <- Rn + 2

operand = (OA)
```

The assembler notation for autoincrement deferred mode is @(Rn)+. For absolute mode the notation is @#address where address is the word which follows the instruction.

10.2.1.5 Autodecrement Mode -

The addressing format for autodecrement mode is:

```
 5   3 2   0
+-----+-----+
|  4  | Rn  |
+-----+-----+
```

In autodecrement mode addressing, the size of the operand in bytes (1 for byte; 2 for word) is subtracted from the contents of register n (except in the case of SP and PC), and the register is replaced by the result. If Rn denotes SP or PC, the register is decremented by 2 and the register is replaced by the result. The updated contents of register n is the address of the operand:

```
if n LEQ 5 then Rn <- Rn - size else Rn <- Rn - 2

OA = Rn

operand = (OA)
```

The assembler notation for autodecrement mode is -(Rn).

10.2.1.6 Autodecrement Deferred Mode -

The addressing format for autodecrement deferred mode is:

```
 5   3 2   0
+-----+-----+
|  5  | Rn  |
+-----+-----+
```

In autodecrement deferred mode addressing, 2 is subtracted from the contents of register n, and the register is replaced by the result. The updated contents of register n is the address of the word whose contents is the address of the operand:

```
        Rn <- Rn - 2

        OA = (Rn)

        operand = (OA)
```

The assembler notation for autodecrement deferred mode is @-(Rn).

1Ø.2.1.7 Index Mode -

The addressing format for index mode is:

```
         5   3 2   Ø
        +-----+-----+
        |  6  | Rn  |
        +-----+-----+
```

In index mode, the index (contents of the word following the instruction) is added to the contents of register n. The result is the address of the operand:

```
        OA = Rn + index

        operand = (OA)
```

If Rn denotes PC, the updated contents of the PC is used, and the mode is termed relative mode.

The assembler notation for index mode is index(Rn), where the index value is the word following the instruction.

1Ø.2.1.8 Index Deferred Mode -

The addressing format for index deferred mode is:

```
         5   3 2   Ø
        +-----+-----+
        |  7  | Rn  |
        +-----+-----+
```

In index deferred mode, the index (contents of the word following the instruction) is added to the contents of register n. The result is the address of a word whose contents is the address of the operand:

```
        OA = (Rn + index)

        operand = (OA)
```

If Rn denotes PC, the updated contents of the PC is used, and the mode is termed relative deferred mode.

The assembler notation for index deferred mode is @index(Rn), where the index value is the word following the instruction.

10.2.2 The Stack

General register R6 is used as the stack pointer by certain instructions, as in the PDP-11. It is not, however, used by the hardware for any exceptions or interrupts. There is also no stack overflow protection in compatibility mode.

10.2.3 Processor Status Word

PDP-11 compatibility mode uses a subset of the full PDP-11 Processor Status Word. The format of the compatibility mode PSW is:

```
 1
 5                    5 4 3 2 1 0
+--------------------+-+-+-+-+-+
|          0         |T|N|Z|V|C|
+--------------------+-+-+-+-+-+
```

When an RTI or RTT instruction is executed, bits 15 through 5 in the saved PSW on the stack are ignored.

10.2.4 Instructions

Table 10.1 lists the instructions provided in compatibility mode.

TABLE 10.1
Compatibility Mode Instructions

Opcode (octal)	Mnemonic
000002	RTI
000006	RTT
0001DD	JMP
00020R	RTS
000240-000277	Condition codes
0003DD	SWAB
000400-003777	Branches
100000-103777	Branches
004RDD	JSR
.050DD	CLR(B)
.051DD	COM(B)
.052DD	INC(B)
.053DD	DEC(B)
.054DD	NEG(B)
.055DD	ADC(B)
.056DD	SBC(B)
.057SS	TST(B)
.060DD	ROR(B)
.061DD	ROL(B)
.062DD	ASR(B)
.063DD	ASL(B)
0065SS	MFPI*
0066DD	MTPI*
1065SS	MFPD*
1066DD	MTPD*
0067DD	SXT
070RSS	MUL
071RSS	DIV
072RSS	ASH
073RSS	ASHC
074RDD	XOR
077RNN	SOB
.1SSDD	MOV(B)
.2SSSS	CMP(B)
.3SSSS	BIT(B)
.4SSDD	BIC(B)
.5SSDD	BIS(B)
06SSDD	ADD
16SSDD	SUB

R = register specifier
SS = source operand specifier
DD = destination operand specifier
. = 0 for word operations and 1 for byte operations

* These instructions execute exactly as they would on a PDP-11 in user mode with Instruction and Data space overmapped. More specifically, they ignore the previous access level and act like PUSH and POP instructions referencing the current stack.

Table 1Ø.2 lists the trap instructions that cause the machine to fault to VAX mode, where either the complete trap may be serviced, or where the instruction may be simulated.

TABLE 1Ø.2
Compatibility Mode Trap Instructions

Opcode (octal)	Mnemonic
ØØØØØ3	BPT
ØØØØØ4	IOT
1Ø4ØØØ-1Ø4377	EMT
1Ø44ØØ-1Ø4777	TRAP

The instructions listed in Table 1Ø.3 and all other opcodes not listed in Tables 1Ø.1 or 1Ø.2 are considered reserved instructions in compatibility mode, and fault to VAX mode. See Section 1Ø.5.

TABLE 1Ø.3
Compatibility Mode Reserved Instructions

Opcode (octal)	Mnemonic
ØØØØØØ	HALT
ØØØØØ1	WAIT
ØØØØØ5	RESET
ØØØØØ7	MFPT
ØØØ23N	SPL
ØØ64NN	MARK
ØØ7ØDD	CSM
Ø75ØØR	FADD--FIS
Ø75Ø1R	FSUB--FIS
Ø75Ø2R	FMUL--FIS
Ø75Ø3R	FDIV--FIS
Ø76XXX	Extended Instructions
1Ø64SS	MTPS
1Ø67DD	MFPS
17XXXX	FP11 Floating Point

Note that no floating point instructions are included in compatibility mode.

10.2.4.1 Single Operand Instructions -

Arithmetic and Logical:

CLR	DEC	INC	NEG	TST	COM
CLRB	DECB	INCB	NEGB	TSTB	COMB

Shifts:

ASR	ASL
ASRB	ASLB

Multiprecision:

ADC	SBC	SXT
ADCB	SBCB	

Rotates:

ROL	ROR	SWAB
ROLB	RORB	

CLR Clear

Format:

```
  1
  5               6 5         Ø
+-----------------+-----------+
|     Opcode      |  dst.wx   |
+-----------------+-----------+
```

Operation:

```
dst <- Ø;
```

Condition Codes:

```
N <- Ø;
Z <- 1;
V <- Ø;
C <- Ø;
```

Exceptions:

none

Opcodes (octal):

ØØ5Ø	CLR	Clear Word
1Ø5Ø	CLRB	Clear Byte

Description:

The destination operand is replaced by zero.

DEC Decrement

Format:

```
 1
 5                 6 5         0
+------------------+-----------+
|      Opcode      |  dst.mx   |
+------------------+-----------+
```

Operation:

```
dst <- dst - 1;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- C;
```

Exceptions:

none

Opcodes (octal):

0053	DEC	Decrement Word
1053	DECB	Decrement Byte

Description:

One is subtracted from the destination operand and the destination operand is replaced by the result.

Note:

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

INC   Increment

Format:

```
 1
 5                  6 5          0
+------------------+------------+
|      Opcode      |  dst.mx    |
+------------------+------------+
```

Operation:

```
dst <- dst + 1;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- C;
```

Exceptions:

none

Opcodes (octal):

0052	INC	Increment Word
1052	INCB	Increment Byte

Description:

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

Note:

Integer overflow occurs if the largest positive integer is incremented. On overflow, the destination operand is replaced by the largest negative integer.

NEG Negate

Format:

```
 1
 5                 6 5         Ø
+-----------------+-----------+
|     Opcode      |  dst.mx   |
+-----------------+-----------+
```

Operation:

```
dst <- -dst;
```

Condition Codes:

```
N <- dst LSS Ø;
Z <- dst EQL Ø;
V <- dst EQL most negative integer;
C <- dst NEQ Ø;
```

Exceptions:

none

Opcodes (octal):

ØØ54	NEG	Negate Word
1Ø54	NEGB	Negate Byte

Description:

The destination operand is negated (2's complement) and the destination operand is replaced by the result.

Note:

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

TST   Test

Format:

```
        1
        5                  6 5          0
        +-------------------+-----------+
        |       Opcode      |  src.rx   |
        +-------------------+-----------+
```

Operation:

```
        src - 0;
```

Condition Codes:

```
        N <- src LSS 0;
        Z <- src EQL 0;
        V <- 0;
        C <- 0;
```

Exceptions:

none

Opcodes (octal):

0057	TST	Test Word
1057	TSTB	Test Byte

Description:

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

COM   Complement

Format:

```
 1
 5                 6 5         0
+-----------------+-----------+
|     Opcode      |  dst.mx   |
+-----------------+-----------+
```

Operation:

```
dst <- NOT dst;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- 1;
```

Exceptions:

none

Opcodes (octal):

0051	COM	Complement Word
1051	COMB	Complement Byte

Description:

The destination operand is complemented (1's complement) and the destination operand is replaced by the result.

ASR Arithmetic Shift Right

Format:

```
 1
 5                 6 5         0
+-----------------+-----------+
|      Opcode     |  dst.mx   |
+-----------------+-----------+
```

Operation:

```
dst <- dst shifted one place to the right;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {bit shifted out} XOR {dst LSS 0};
C <- bit shifted out;
```

Exceptions:

none

Opcodes (octal):

0062	ASR	Arithmetic Shift Right Word
1062	ASRB	Arithmetic Shift Right Byte

Description:

The destination operand is arithmetically shifted right by one bit and the destination operand is replaced by the result.

Notes:

1. The sign bit of the destination operand is replicated in shifts to the right. The condition code C bit stores the bit shifted out.

2. If the PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.

ASL   Arithmetic Shift Left

Format:

```
 1
 5               6 5         0
+-----------------+-----------+
|     Opcode      |  dst.mx   |
+-----------------+-----------+
```

Operation:

```
dst <- dst shifted one place to the left;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- bit shifted out;
```

Exceptions:

none

Opcodes (octal):

0063	ASL	Arithmetic Shift Left Word
1063	ASLB	Arithmetic Shift Left Byte

Description:

The destination operand is arithmetically shifted left by one bit and the destination operand is replaced by the result.

Notes:

1. The least significant bit is filled with zero in shifts to the left. The condition code C bit stores the bit shifted out.

2. Integer overflow occurs if the destination changes sign due to the shift.

ADC Add Carry

Format:

```
 1
 5                  6 5          0
+------------------+------------+
|      Opcode      |  dst.mx    |
+------------------+------------+
```

Operation:

```
dst <- dst + C;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- {carry from most significant bit};
```

Exceptions:

none

Opcodes (octal):

0055	ADC	Add Carry to Word
1055	ADCB	Add Carry to Byte

Description:

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

Note:

Integer overflow occurs if the most positive integer is incremented. On overflow, the result is the most negative integer.

SBC   Subtract Carry

Format:

```
 1
 5                 6 5          0
+-----------------+------------+
|      Opcode     |   dst.mx   |
+-----------------+------------+
```

Operation:

```
dst <- dst - C;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- {borrow into most significant bit};
```

Exceptions:

none

Opcodes (octal):

0056	SBC	Subtract Carry from Word
1056	SBCB	Subtract Carry from Byte

Description:

The contents of the condition code C bit are subtracted from the destination operand and the destination operand is replaced by the result.

Note:

Integer overflow occurs if the most negative integer is decremented. On overflow, the result is the most positive integer.

SXT   Sign Extend Word

Format:

```
 1
 5                  6 5          0
+------------------+------------+
|      Opcode      |  dst.ww    |
+------------------+------------+
```

Operation:

```
if N EQL 1 then dst <- -1 else dst <- 0;
```

Condition Codes:

```
N <- dst LSS 0; !N <- N
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes (octal):

```
0067    SXT Sign Extend
```

Description:

If the condition code N bit is set then the destination operand is replaced by -1; otherwise the destination operand is cleared.

Note:

If the PC is used as the destination operand, the results and the next instruction executed are UNPREDICTABLE.

ROL   Rotate Left

Format:

```
 1
 5                 6 5         0
+-----------------+-----------+
|      Opcode     |  dst.mx   |
+-----------------+-----------+
```

Operation:

```
dst'C <- dst'C rotated left;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- {bit rotated out of dst};
```

Exceptions:

none

Opcodes (octal):

0061	ROL	Rotate Left Word
1061	ROLB	Rotate Left Byte

Description:

The condition code C bit and the destination operand are rotated left by one bit position; i.e. the C bit gets the most significant bit of the destination operand, the destination is replaced by the destination shifted left by one bit with the initial C bit filling the least significant bit.

Notes:

1. The rotate instructions operate on the destination operand and the condition code C bit taken as a circular datum.

2. Integer overflow occurs if the destination changes sign due to the rotate.

ROR   Rotate Right

Format:

```
 1
 5                    6 5          0
+--------------------+------------+
|       Opcode       |  dst.mx    |
+--------------------+------------+
```

Operation:

```
dst'C <- dst'C rotated right;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {C bit changed due to rotate};
C <- {bit rotated out of dst};
```

Exceptions:

none

Opcodes (octal):

0060	ROR	Rotate Right Word
1060	RORB	Rotate Right Byte

Description:

The condition code C bit and the destination operand are rotated right by one bit position; i.e. the C bit gets the least significant bit of the destination operand, the destination is replaced by the destination shifted right by one bit with the initial C bit filling the most significant bit.

Note:

The rotate instructions operate on the destination operand and the condition code C bit taken as a circular datum.

SWAB Swap Bytes

Format:

```
 1
 5               6 5        0
+-----------------+----------+
|     Opcode      | dst.mw   |
+-----------------+----------+
```

Operation:

```
dst <- dst<7:0>'dst<15:8>;
```

Condition Codes:

```
N <- dst<7:0> LSS 0;
Z <- dst<7:0> EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes (octal):

0003	SWAB	Swap Bytes

Description:

The high and low bytes of the destination word operand are swapped.

Note:

If the PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.

10.2.4.2 Double Operand Instructions -

Arithmetic and Logical:

MOV	ADD	SUB	CMP	MUL	DIV	XOR	BIS	BIC	BIT
MOVB			CMPB				BISB	BICB	BITB

Shift:

ASH ASHC

If a register that is used in the source operand specifier in autoincrement or autodecrement modes is also used in the destination (or source 2) operand specifier, the updated value of the register is used to evaluate the destination specifier. Side effects caused by a destination address calculation have no effect on source values.

MOV Move

Format:

```
 1     1 1
 5     2 1         6 5         Ø
+-------+----------+-----------+
|Opcode |  src.rx  |  dst.wx   |
+-------+----------+-----------+
```

Operation:

```
dst <- src;
```

Condition Codes:

```
N <- dst LSS Ø;
Z <- dst EQL Ø;
V <- Ø;
C <- C;
```

Exceptions:

none

Opcodes (octal):

Ø1	MOV	Move Word
11	MOVB	Move Byte

Description:

The destination operand is replaced by the source operand.

Note:

The low byte is sign-extended on a MOVB to a register;i.e. bits <15:8> of the destination register are replaced by bit <7> of the source operand.

ADD Add

Format:

```
 1     1 1          6 5         0
 5     2 1
+-------+-----------+-----------+
|Opcode |  src.rw   |  dst.mw   |
+-------+-----------+-----------+
```

Operation:

```
dst <- dst + src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- {carry from most significant digit};
```

Exceptions:

none

Opcodes (octal):

06	ADD	Add Word

Description:

The source operand is added to the destination operand and the destination operand is replaced by the result.

Note:

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

SUB   Subtract

Format:

```
 1     1 1
 5     2 1         6 5         0
+-------+-----------+-----------+
|Opcode |  src.rw   |  dst.mw   |
+-------+-----------+-----------+
```

Operation:

```
dst <- dst - src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- {borrow into most significant digit};
```

Exceptions:

none

Opcodes (octal):

16	SUB	Subtract Word

Description:

The source operand is subtracted from the destination operand and the destination operand is replaced by the result.

Note:

Integer overflow occurs if the input operands are of different signs and the result has the sign of the source. On overflow, the destination operand is replaced by the low order bits of the true result.

CMP   Compare

Format:

```
 1      1 1
 5      2 1         6 5         0
+-------+-----------+-----------+
|Opcode |  src1.rx  |  src2.rx  |
+-------+-----------+-----------+
```

Operation:

```
tmp <- src1 - src2;
```

Condition Codes:

```
N <- tmp LSS 0;
Z <- tmp EQL 0;
V <- {integer overflow};
C <- {borrow into most significant digit};
```

Exceptions:

none

Opcodes (octal):

02	CMP	Compare Word
12	CMPB	Compare Byte

Description:

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

Note:

Integer overflow occurs if the operands are of different sign and the result of the subtraction (src1 - src2) has the same sign as the source 2 operand.

MUL Multiply

Format:

```
 1
 5           9 8   6 5         0
+------------+-----+-----------+
|   Opcode   | reg |  src.rw   |
+------------+-----+-----------+
```

Operation:

```
tmp<31:0> <- Rn * src;
Rn <- tmp<31:16>;
R[n OR 1] <- tmp<15:0>;
```

Condition Codes:

```
N <- tmp LSS 0;
Z <- tmp EQL 0;
V <- 0;
C <- {result unrepresentable in 16 bits};
```

Exceptions:

none

Opcodes (octal):

070	MUL	Multiply Word

Description:

The destination register is multiplied by the source operand. The most significant 16 bits of the 32-bit product are stored in register Rn. Then the least significant 16 bits are stored in R[n OR 1]. The condition codes are set based on the 32-bit result.

Note:

1. The C bit is set if the result of the multiplication cannot be represented in 16 bits; i.e. the 32-bit product is less than -2**15 or greater than or equal to 2**15.

2. If an odd numbered register is used as the destination, the low order sixteen bits are stored as the result.

3. If R6 or PC is used as the destination, the next instruction executed and the result are UNPREDICTABLE.

DIV   Divide

Format:

```
        1
        5           9 8   6 5         0
        +-------------+-----+-----------+
        |    Opcode   | reg |  src.rw   |
        +-------------+-----+-----------+
```

Operation:

```
        tmp <- Rn'R[n OR 1]
        Rn <- tmp / src;
        R[n OR 1] <- REM(tmp , src);
```

Condition Codes:

```
        N <- Rn LSS 0;   !UNPREDICTABLE if V is set
        Z <- Rn EQL 0;   !UNPREDICTABLE if V is set
        V <- {src EQL 0} OR {integer overflow};
        C <- {src EQL 0};
```

Exceptions:

none

Opcodes (octal):

071	DIV	Divide

Description:

If the source operand is not zero, the 32-bit integer in Rn'R[n OR 1] is divided by the source operand. The quotient is stored in Rn, and the remainder is stored in R[n OR 1]. The remainder has the same sign as the dividend. If the source operand is zero, the instruction terminates without modifying the destination registers.

Notes:

1. Integer overflow occurs if the quotient is less than -2**15 or greater than or equal to 2**15. On integer overflow, the contents of the destination registers are UNPREDICTABLE.

2. If an odd register or R6 is used as the destination, the results are UNPREDICTABLE. Furthermore, if R6 or PC is used as the destination, the next instruction executed is UNPREDICTABLE.

XOR   Exclusive OR

Format:

```
 1
 5              9 8   6 5         0
+--------------+-----+------------+
|    Opcode    | reg |  dst.mw    |
+--------------+-----+------------+
```

Operation:

```
dst <- Rn XOR dst;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes (octal):

074	XOR	Exclusive OR Word

Description:

The source register is XORed with the destination operand and the destination operand is replaced by the result.

BIS Bit Set

Format:

```
         1      1 1
         5      2 1          6 5          0
        +-------+-----------+-----------+
        |Opcode |  src.rx   |  dst.mx   |
        +-------+-----------+-----------+
```

Operation:

```
        dst <- dst OR src;
```

Condition Codes:

```
        N <- dst LSS 0;
        Z <- dst EQL 0;
        V <- 0;
        C <- C;
```

Exceptions:

none

Opcodes (octal):

05	BIS	Bit Set Word
15	BISB	Bit Set Byte

Description:

The source operand is ORed with the destination operand and the destination operand is replaced by the result.

BIC   Bit Clear

Format:

```
  1     1 1
  5     2 1         6 5         0
 +-------+-----------+-----------+
 |Opcode |  src.rx   |  dst.mx   |
 +-------+-----------+-----------+
```

Operation:

```
dst <- dst AND {NOT src};
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes (octal):

04	BIC	Bit Clear Word
14	BICB	Bit Clear Byte

Description:

The destination operand is ANDed with the 1's complement of the source operand and the destination operand is replaced by the result.

BIT   Bit Test

Format:

```
 1      1 1
 5      2 1          6 5          0
+-------+-----------+-----------+
|Opcode |  src1.rx  |  src2.rx  |
+-------+-----------+-----------+
```

Operation:

```
tmp <- src1 AND src2;
```

Condition Codes:

```
N <- tmp LSS 0;
Z <- tmp EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes (octal):

03	BIT	Bit Test Word
13	BITB	Bit Test Byte

Description:

The source 1 operand is ANDed with the source 2 operand. The only action is to set the condition codes.

ASH   Arithmetic Shift

Format:

```
 1
 5          9 8   6 5         Ø
+-----------+-----+-----------+
|   Opcode  | reg |  src.rw   |
+-----------+-----+-----------+
```

Operation:

```
Rn <- Rn shifted src<5:Ø> bits;
```

Condition Codes:

```
N <- Rn LSS Ø;
Z <- Rn EQL Ø;
V <- if src<5:Ø> EQL Ø then Ø else {integer overflow};
C <- if src<5:Ø> EQL Ø then Ø else {last bit shifted out};
```

Exceptions:

none

Opcodes (octal):

Ø72   ASH   Arithmetic Shift

Description:

The specified register is arithmetically shifted by the number of bits specified by the count operand (bits <5:Ø> of the source operand) and the register is replaced by the result. The count ranges from -32 to +31. A negative count signifies a right shift. A positive count signifies a left shift. A zero count implies no shift; but condition codes are affected.

Notes:

1. The sign bit of Rn is replicated in shifts to the right. The least significant bit is filled with zero in shifts to the left. The C bit stores the last bit shifted out.

2. Integer overflow occurs on a left shift if any bit shifted into the sign position differs from the initial sign bit of the register.

3. If the PC is used as the destination operand the result and the next instruction executed are UNPREDICTABLE.

ASHC Arithmetic Shift Combined

Format:

```
 1
 5           9 8   6 5         0
+-------------+-----+-----------+
|   Opcode    | reg |  src.rw   |
+-------------+-----+-----------+
```

Operation:

```
tmp <- Rn'R[n OR 1];
tmp <- tmp shifted src<5:0> bits;
Rn <- tmp<31:16>;
R[n OR 1] <- tmp<15:0>;
```

Condition Codes:

```
N <- tmp LSS 0;
Z <- tmp EQL 0;
V <- if src<5:0> EQL 0 then 0 else {integer overflow};
C <- if src<5:0> EQL 0 then 0 else {last bit shifted out};
```

Exceptions:

none

Opcodes (octal):

073	ASHC	Arithmetic Shift Combined

Description:

The contents of the specified register, Rn, and the register R[n OR 1] are treated as a single 32-bit operand and are shifted by the number of bits specified by the count operand (bits <5:0> of the source operand) and the registers are replaced by the result. First, bits <31:16> of the result are stored in register Rn. Then, bits <15:0> of the result are stored in register R[n OR 1]. The count ranges from -32 to +31. A negative count signifies a right shift. A positive count signifies a left shift. A zero count implies no shift; but condition codes are affected. Condition codes are always set on the 32-bit result.

Notes:

1. The sign bit of Rn is replicated in shifts to the right. The least significant bit is filled with zero in shifts to the left. The C bit stores the last bit shifted out.

2. Integer overflow occurs on a left shift if any bit shifted into the sign position differs from the initial sign bit of the 32-bit operand.

3. If the SP or PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.

10.2.4.3 Branch Instructions -

BR	BNE	BPL	BVC	BCC	BGE	BGT	BHI	BHIS	SOB
	BEQ	BMI	BVS	BCS	BLT	BLE	BLOS	BLO	

BR   Branch

Format:

```
  1
  5               8 7              0
  +----------------+----------------+
  |     Opcode     |    displ.bb    |
  +----------------+----------------+
```

Operation:

```
PC <- PC + SEXT(2*displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes (octal):

0004	BR	Branch

Description:

Twice the sign-extended displacement is added to the PC and the PC is replaced by the result.

B Branch on (condition)

Format:

```
 1
 5              8 7              0
+---------------+----------------+
|     Opcode    |    displ.bb    |
+---------------+----------------+
```

Operation:

```
        if condition then PC <- PC + SEXT(2*displ);
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

none

Opcodes (octal):

		Condition	
0014	BEQ	Z EQL 1	Branch on Equal
0010	BNE	Z EQL 0	Branch Not Equal
1004	BMI	N EQL 1	Branch on Minus
1000	BPL	N EQL 0	Branch on Plus
1034	BCS, BLO	C EQL 1	Branch on Carry Set, Branch on Lower
1030	BCC, BHIS	C EQL 0	Branch on Carry Clear, Branch on Higher or Same
1024	BVS	V EQL 1	Branch on Overflow Set
1020	BVC	V EQL 0	Branch on Overflow Clear
0024	BLT	{N XOR V} EQL 1	Branch on Less Than
0020	BGE	{N XOR V} EQL 0	Branch on Greater Than or Equal
0034	BLE	{Z OR {N XOR V}} EQL 1	Branch on Less Than or Equal
0030	BGT	{Z OR {N XOR V}} EQL 0	Branch on Greater Than
1010	BHI	{C OR Z} EQL 0	Branch on Higher
1014	BLOS	{C OR Z} EQL 1	Branch on Lower or Same

Description:

The condition codes are tested and if the condition indicated by the instruction is met, twice the sign-extended displacement is added to the PC and the PC is replaced by the result.

SOB Subtract One and Branch

Format:

```
 1
 5          9 8   6 5         0
+------------+-----+-----------+
|   Opcode   | reg | displ.b6  |
+------------+-----+-----------+
```

Operation:

```
Rn <- Rn - 1;
if Rn NEQ 0 then PC <- PC - ZEXT(2*displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes (octal):

```
077     SOB     Subtract One and Branch
```

Description:

One is subtracted from the specified register and the register is replaced by the result. If the register is not equal to zero, twice the zero-extended displacement is subtracted from the PC and the PC is replaced by the result.

Notes:

1. If the PC is specified as the register, the results and the next instruction executed are UNPREDICTABLE.

2. The 6-bit displacement operand is contained in bits <5:0> of the instruction.

10.2.4.4 Jump And Subroutine Instructions -

```
JMP     JSR
        RTS
```

JMP   Jump

Format:

```
        1
        5                  6 5          0
       +-------------------+------------+
       |      Opcode       |  dst.aw    |
       +-------------------+------------+
```

Operation:

```
        PC <- dst;
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

compatibility mode illegal instruction

Opcodes (octal):

```
        0001    JMP     Jump
```

Description:

The PC is replaced by the destination operand.

Note:

A compatibility mode illegal instruction fault occurs if destination mode 0 is used.

JSR Jump to Subroutine

Format:

```
 1
 5          9 8   6 5        0
+-----------+-----+----------+
|   Opcode  | reg |  dst.aw  |
+-----------+-----+----------+
```

Operation:

```
tmp <- dst;
-(SP) <- Rn;    !value of Rn affected by dst specifier evaluation
Rn <- PC;
PC <- tmp;
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

compatibility mode illegal instruction

Opcodes (octal):

004	JSR	Jump to Subroutine

Description:

The source register is pushed on the stack and the source register is replaced by the PC. The PC is replaced by the destination operand.

Notes:

1. A compatibility mode illegal instruction fault occurs if destination mode 0 is used.

2. If the destination uses the same register as the source in the autoincrement or autodecrement addressing modes, the updated contents of the register are pushed on the stack.

RTS Return from Subroutine

Format:

```
         1
         5                        3 2   0
        +-------------------------+-----+
        |         Opcode          | reg |
        +-------------------------+-----+
```

Operation:

```
        PC <- Rn;
        Rn <- (SP)+;
```

Condition Codes:

```
        N <- N;
        Z <- Z;
        V <- V;
        C <- C;
```

Exceptions:

none

Opcodes (octal):

00020	RTS	Return from subroutine

Description:

The PC is replaced by the destination register. The destination register is replaced by a word popped from the stack.

1Ø.2.4.5 Return From Interrupts And Traps -

RTI RTT

RTI	Return from Interrupt
RTT	Return from Trap

Format:

```
 1
 5                               0
+--------------------------------+
|            Opcode              |
+--------------------------------+
```

Operation:

```
PC <- (SP)+;
PSW<4:0> <- {(SP)+}<4:0>;
```

Condition Codes:

```
N <- saved PSW<3>;
Z <- saved PSW<2>;
V <- saved PSW<1>;
C <- saved PSW<0>;
```

Exceptions:

none

Opcodes (octal):

000002	RTI	Return from Interrupt
000006	RTT	Return from Trap

Description:

The PC is replaced by the first word popped from the stack. The low 5 bits of the PSW are replaced by the corresponding bits of the second word popped from the stack.

Notes:

1. In compatibility mode, the RTI and RTT instructions ignore the high 11 bits of the PSW popped from the stack.

2. In compatibility mode, the RTI and RTT instructions are identical.

10.2.4.6 Miscellaneous -

MTPI	MTPD	SCC
MFPI	MFPD	CCC

MTP   Move To Previous Space

Format:

```
 1
 5                     6 5          0
+--------------------+------------+
|       Opcode       |  dst.ww    |
+--------------------+------------+
```

Operation:

```
dst <- (SP)+;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes (octal):

0066	MTPI	Move To Previous Instruction Space
1066	MTPD	Move To Previous Data Space

Description:

In compatibility mode, this PDP-11 instruction works like a POP instruction. The destination operand is replaced by a word popped from the stack.

Note:

The implied source operand specifier is evaluated before the destination specifier.

MFP   Move From Previous Space

Format:

```
 1
 5                  6 5          0
+------------------+-----------+
|      Opcode      | src.rw    |
+------------------+-----------+
```

Operation:

```
-(SP) <- src;
```

Condition Codes:

```
N <- src LSS 0;
Z <- src EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes (octal):

0065	MFPI	Move From Previous Instruction Space
1065	MFPD	Move From Previous Data Space

Description:

In compatibility mode, this PDP-11 instruction works like a PUSH instruction. The source operand is pushed onto the stack.

CC   Condition Code Operators

Format:

```
         1
         5                    5 4       0
        +--------------------+---------+
        |       Opcode       |  mask   |
        +--------------------+---------+
```

Operation:

```
        if mask<4> EQL 1 then PSW<3:0> <- PSW<3:0> OR mask<3:0>
                else PSW<3:0> <- PSW<3:0> AND {NOT mask<3:0>};
```

Condition Codes:

```
        if mask<4> EQL 1 then
                begin
                N <- N OR mask<3>;
                Z <- Z OR mask<2>;
                V <- V OR mask<1>;
                C <- C OR mask<0>;
                end
        else
                begin
                N <- N AND {NOT mask<3>};
                Z <- Z AND {NOT mask<2>};
                V <- V AND {NOT mask<1>};
                C <- C AND {NOT mask<0>};
                end
```

Exceptions:

none

Opcodes (octal):

000240		No operation
000241	CLC	Clear C
000242	CLV	Clear V
000244	CLZ	Clear Z
000250	CLN	Clear N
000257	CCC	Clear all Condition Codes
000261	SEC	Set C
000262	SEV	Set V
000264	SEZ	Set Z
000270	SEN	Set N
000277	SCC	Set all Condition Codes

Combinations of the above set or clear operations may be

ORed together to form combined instructions.

Description:

If the mask<4> bit is set, the PSW condition code bits are ORed with mask<3:Ø> and the condition codes are replaced by the result. If the mask<4> bit is clear, the PSW condition code bits are ANDed with the 1's complement of mask<3:Ø> and the condition codes are replaced by the result.

10.3 ENTERING AND LEAVING COMPATIBILITY MODE

Compatibility mode is entered by executing an REI instruction with the compatibility mode bit set in the image of the PSL on the stack. Other bits in the PSL have the following effects:

Bits	Effect
NZVC	Condition Codes
T	T Bit
DV	Reserved operand fault if not zero
FU	Reserved operand fault if not zero
IV	Reserved operand fault if not zero
IPL	Reserved operand fault if not zero
PRV MOD	Reserved operand fault if not 3
CUR MOD	Reserved operand fault if not 3
IS	Reserved operand fault if not zero
FPD	Reserved operand fault if not zero
TP	T pending bit. See Section on T bit operation in compatibility mode for a complete description of how trace faults work in compatibility mode.

VAX native mode is re-entered from compatibility mode by the compatibility mode program causing an exception, or by an interrupt. The PSL pushed on the kernel or interrupt stack when leaving compatibility mode has all the bits that cause reserved operand faults in the above table set to the appropriate state.

Note that when an RTI or RTT instruction is executed in compatibility mode, the 11 high bits of the PSW are ignored, but when the PSW is restored as part of the PSL when going from VAX native mode to compatibility mode, those bits must be zero, or a reserved operand fault occurs.

10.3.1 General Register Usage

Compatibility mode registers R0 through R6 are bits 15 through 0 of VAX general registers R0 through R6, respectively. Compatibility mode register R7 (PC) is bits 15 through 0 of VAX general register R15 (PC). VAX registers R8 through R14 (SP) are not affected by compatibility mode. When entering compatibility mode, VAX register R7 and the upper halves of registers R0 through R6 and R15 are ignored. When an exception or interrupt occurs from compatibility mode, VAX register R7 is UNPREDICTABLE and the upper halves of R0 through R6 are either cleared or left unchanged. and the upper half of the stacked R15 (PC) is zero. Since there are no FP11 floating point instructions in compatibility mode, there are no floating accumulators.

10.4 COMPATIBILITY MODE MEMORY MANAGEMENT

PDP-11 addresses are 16-bit byte addresses, hence compatibility mode programs are confined to execute in the first 64k bytes of the per process part of the virtual address space. There is a one-to-one correspondence between a compatibility mode virtual address and its VAX counterpart (e.g., virtual address 0 references the same location in both modes). A compatibility mode address is interpreted as follows:

```
 31                              16 15          9 8               0
+--------------------------------+-------------+-----------------+
|               0                |    PAGE     |  DISPLACEMENT   |
+--------------------------------+-------------+-----------------+
```

PDP-11 segments can consist of 1 to 128 blocks of 64 bytes. VAX pages are 512 bytes long. The PDP-11 capability of providing different access protection to different segments is provided in 8 block chunks since protection is specified at the page level in the VAX architecture.

The memory management system protects and relocates compatibility mode addresses in the normal native mode manner. Thus, all of the memory management mechanisms available in VAX mode are available to the compatibility mode executive for managing both the virtual and physical memory of compatibility mode programs. All of the exception conditions that can be caused by memory management in VAX mode can also occur when relocating a compatibility mode address. See Chapter 5.

Most of the KT-11 features that affect the user environment can be simulated with the VAX memory management system. Table 10-4 briefly describes the simulation method. Refer to Chapter 5 of this manual and the appropriate PDP-11 documents for details of each system.

Table 10-4

KT11-D feature to be simulated	VAX simulation method
8 segments per user.	8 segments can be simulated by dividing the 128 pages of the compatibility mode virtual address spac into 8 logical groups of 16 pages each having possibly different protection.
Segment size from 64 to 8K bytes (1 to 128 blocks) in 64 byte increments, using contiguous memory.	Segment size from 512 to 8K bytes (1 to 16 pages) in 512 byte (1 page) increments, using discontiguous memory.
Forward growing segments (Expand Direction=0).	Can be simulated using page table entries specifying no access for those pages that are not allocated.
Backward growing segments (ED=1).	Can be simulated using page table entries specifying no access for those pages that are not allocated.
Segments begin on any 64 byte boundary.	Segments begin on any 512 byte boundary.

The following example shows how a PDP-11 environment can be simulated using VAX memory management. Segments 0, 1, and 2 of the PDP-11 environment are program segments; 3 is unused; 4 and 5 are stack; and 6 and 7 are read-write data.

11 Environment				VAX Page Table	
Seg #	Size (bytes)	Expand Direction	Access	Page	Access
0	8K	Up	Read only	0-15	Read only
1	8K	Up	Read only	16-31	Read only
2	256	Up	Read only	32	Read only
3	0	--	None	33-77	No Access
4	1K	Down	Read-Write	78-79	Read-Write
5	8K	Down	Read-Write	80-95	Read-Write
6	8K	Up	Read-Write	96-111	Read-Write
7	2K	Up	Read-Write	112-115	Read-Write
				116-127	No Access

10.5 COMPATIBILITY MODE EXCEPTIONS AND INTERRUPTS

All interrupts and exception conditions that occur while the machine is in compatibility mode cause the machine to enter VAX mode, and are serviced as indicated in Chapter 6 (note that this includes backing up instruction side effects if necessary). The exception conditions discussed in this section are specific to compatibility mode. All these exceptions create a 3-longword frame on the kernel stack containing PSL, PC, and one longword of exception specific information. Bits 15 through 0 of this longword contain a code indicating the specific type of exception and bits 31 through 16 are zero. There are no compatibility mode exception conditions that result in traps (see Chapter 6 for definition of trap, fault, and abort).

10.5.1 Reserved Instruction Fault

A reserved instruction fault occurs for opcodes that are defined as reserved in compatibility mode (see section on Instructions). The code for the reserved instruction fault is 0.

10.5.2 BPT Instruction Fault

The code for the BPT instruction fault is 1.

10.5.3 IOT Instruction Fault

The code for the IOT instruction fault is 2.

10.5.4 EMT Instruction Fault

The fault code for the group of EMT instructions is 3.

10.5.5 TRAP Instruction Fault

The fault code for the group of TRAP instructions is 4.

10.5.6 Illegal Instruction Fault

In compatibility mode,JMP and JSR instructions with a register destination are illegal. The fault code for illegal instructions is 5.

10.5.7 Odd Address Error Abort

An odd address error abort is caused in compatibility mode whenever a word reference is attempted on a byte boundary. The code for odd address errors is 6.

10.6 T BIT OPERATION IN COMPATIBILITY MODE

In compatibility mode, a trace fault occurs at the beginning of an instruction when the T bit is set in the PSW at the beginning of the prior instruction. This effect is achieved by using the TP bit in the PSL (see Chapter 6). On trace faults, a 2-longword kernel stack frame is created, containing PSL and PC. IPL and IS are zero and CM is one in the stacked PSL. Compatibility mode trace fault uses the same vector as VAX mode trace fault. See Chapter 6. The rules for trace fault generation in compatibility mode are identical to those for native mode. However, an odd address abort for an instruction fetch may precede the trace fault for that instruction.

There are two ways to get the T bit set at the beginning of a compatibility mode instruction:

1. An RTT/RTI instruction is executed in compatibility mode with the T bit is set in the PSW image on the stack. In this case, the next instruction is executed (the one pointed to by the PC on the stack), and a trace fault is taken before the following instruction.

2. An REI instruction is executed in VAX mode which has both the T bit and CM bit set (and TP clear) in the saved PSL image on the stack. Again, one instruction is executed, and the trace fault is taken. (For a complete description of the interaction of REI, T bit, and TP bit, see Chapter 6. The operations that occur as a function of these conditions are the same whether or not compatibility mode is being entered from the REI.)

The T bit interacts with other compatibility mode operations as follows (for interaction with other than compatibility mode specific operations, see Chapter 6):

1. T bit set (but TP is clear) at the beginning of any compatibility mode instruction which does not cause a compatibility mode fault.

   In this case, the instruction sets TP and executes. A trace fault is taken before the next instruction. The saved PSL has the T bit set and TP clear. The compatibility mode executive can take one of the following courses of action:

   1. If it services the exception directly, it can clear the T bit in the saved PSL on the kernel stack if it no longer wants to trace the program, or it can leave it set if it wants to continue tracing the program. It exits with an REI.

   2. If it returns the trace exception to compatibility mode, it pushes a (16-bit) PC and (16-bit) PSW with the T bit set on the compatibility mode User stack to simulate the effect of the PDP-11 trace trap. It then clears the T bit in the saved PSL image on the kernel stack, changes the saved PC to point to the compatibility mode service routine, and does an REI. The compatibility mode service routine can then clear the T bit in the PSW image on its stack if it does not want to continue tracing. The compatibility mode routine returns with RTT or RTI.

2. T bit set (but TP is clear) at the beginning of an RTI or RTT.

   The RTT/RTI instruction executes and TP is set. A trace fault occurs before the next instruction is executed. There are two different cases, depending on whether or not the T bit was set in the image of the PSW which was popped from the stack by the RTT/RTI instruction:

   1. T bit not set.

      Neither TP nor T will be set in the saved PSL on the kernel stack.

   2. T bit set.

      TP will not be set, and T will be set, as is the case as for other compatibility mode instructions.

3. T bit set (but TP is clear) at the beginning of any instruction which causes a compatibility mode fault.

The fault condition is serviced first. TP is clear and T is set in the saved PSL pushed on the kernel stack.

10.7 UNIMPLEMENTED PDP-11 TRAPS

Several traps that occur in PDP-11s are not implemented in compatibility mode:

1. There is no stack overflow trap. This is equivalent to the User Mode of the KT11, where there is also no overflow protection. Stack overflow can be provided by the compatibility mode executive using the memory management mechanisms.

2. There is no concept of a double error trap in compatibility mode, since the first error always puts the machine in VAX mode.

3. All other exception conditions such as power failure, memory parity, and memory management exceptions cause the machine to enter VAX mode.

10.8 COMPATIBILITY MODE I/O REFERENCES

Neither instruction stream references nor data reads nor writes can be to I/O space. The results are UNPREDICTABLE if I/O space is referenced from compatibility mode.

10.9 PROCESSOR REGISTERS

The only processor register available in compatibility mode is part of the PSW, and it maybe explicitly referenced only with the condition code instructions, RTI, and RTT. Access to all other registers must be done in VAX mode.

10.10 PROGRAM SYNCHRONIZATION

All PDP-11s guarantee that read-modify-write operations to I/O device registers are interlocked; that is, the device can determine at the time of the read that the same register will be written as the next bus cycle. This synchronization also works in memory on most PDP-11s. In compatibility mode, instructions that have modify destinations will perform this synchronization for UNIBUS I/O device registers and never for memory.

APPENDIX A
INSTRUCTION SET AND OPCODE ASSIGNMENTS

23-Mar-81 -- Rev 17.1

A.1 INSTRUCTION OPERAND FORMATS

The format of the instructions is given using the qualified name convention described in the next section. For the mnemonics {} encloses a list of data types of which one must be selected. Instructions which have two forms differing in the number of operands have the number of operands appended to the opcode as a digit. For the operands, {} encloses all implied operands. Refer to the VAX-11 Macro Reference Manual for a description of when the data type suffix and operand number suffix may be omitted.

		Instructions
1.	Move MOV{B,W,L,F,D,G,H,Q,O} src.rx, dst.wx	9
2.	Push Long PUSHL src.rl, {-(SP).wl}	1
3.	Clear CLR{B,W,L=F,Q=D=G,O=H} dst.wx	5
4.	Move Negated MNEG{B,W,L,F,D,G,H} src.rx, dst.wx	7
5.	Move Complemented MCOM{B,W,L} src.rx, dst.wx	3
6.	Move Zero-Extended MOVZ{BW,BL,WL} src.rx, dst.wy	3
7.	Convert CVT{B,W,L,F,D,G,H}{B,W,L,F,D,G,H} src.rx, dst.wy All pairs except BB,WW,LL,FF,DD,GG,HH,DG, and GD	4Ø

```
 8.  Convert Rounded                                          4
     CVTR{F,D,G,H}L src.rx, dst.wl

 9.  Compare                                                  7
     CMP{B,W,L,F,D,G,H} src1.rx, src2.rx

10.  Test                                                     7
     TST{B,W,L,F,D,G,H} src.rx

11.  Add 2 Operand                                            7
     ADD{B,W,L,F,D,G,H}2 add.rx, sum.mx

12.  Add 3 Operand                                            7
     ADD{B,W,L,F,D,G,H}3 add1.rx, add2.rx, sum.wx

13.  Increment                                                3
     INC{B,W,L} sum.mx

14.  Add With Carry                                           1
     ADWC add.rl, sum.ml

15.  Add Aligned Word                                         1
     ADAWI add.rw, sum.mw

16.  Subtract 2 Operand                                       7
     SUB{B,W,L,F,D,G,H}2 sub.rx, dif.mx

17.  Subtract 3 Operand                                       7
     SUB{B,W,L,F,D,G,H}3 sub.rx, min.rx, dif.wx

18.  Decrement                                                3
     DEC{B,W,L} dif.mx

19.  Subtract With Carry                                      1
     SBWC sub.rl, dif.ml

20.  Multiply 2 Operand                                       7
     MUL{B,W,L,F,D,G,H}2 mulr.rx, prod.mx

21.  Multiply 3 Operand                                       7
     MUL{B,W,L,F,D,G,H}3 mulr.rx, muld.rx, prod.wx

22.  Extended Multiply                                        1
     EMUL mulr.rl, muld.rl, add.rl, prod.wq

23.  Divide 2 Operand                                         7
     DIV{B,W,L,F,D,G,H}2 divr.rx, quo.mx

24.  Divide 3 Operand                                         7
     DIV{B,W,L,F,D,G,H}3  divr.rx, divd.rx, quo.wx

25.  Extended Divide                                          1
     EDIV divr.rl, divd.rq, quo.wl, rem.wl
```

```
26.  Arithmetic Shift                                            2
     ASH{L,Q} cnt.rb, src.rx, dst.wx

27.  Bit Test                                                    3
     BIT{B,W,L} mask.rx, src.rx

28.  Bit Set 2 Operand                                           3
     BIS{B,W,L}2 mask.rx, dst.mx

29.  Bit Set 3 Operand                                           3
     BIS{B,W,L}3 mask.rx, src.rx, dst.wx

3Ø.  Bit Clear 2 Operand                                         3
     BIC{B,W,L}2 mask.rx, dst.mx

31.  Bit Clear 3 Operand                                         3
     BIC{B,W,L}3 mask.rx, src.rx, dst.wx

32.  Exclusive OR 2 Operand                                      3
     XOR{B,W,L}2 mask.rx, dst.mx

33.  Exclusive OR 3 Operand                                      3
     XOR{B,W,L}3 mask.rx, src.rx, dst.wx

34.  Rotate Long                                                 1
     ROTL cnt.rb, src.rl, dst.wl

35.  Extended Modulus                                            4
     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

36.  Polynomial Evaluation F_floating                            1
     POLYF arg.rf, degree.rw, tbladdr.ab, {RØ-3.wl}

37.  Polynomial Evaluation D_floating                            1
     POLYD arg.rd, degree.rw, tbladdr.ab, {RØ-5.wl}

38.  Polynomial Evaluation G_floating                            1
     POLYG arg.rg, degree.rw, tbladdr.ab, {RØ-5.wl}

39.  Polynomial Evaluation H_floating                            1
     POLYH arg.rh, degree.rw, tbladdr.ab,
     {RØ-5.wl,-16(SP):-1(SP).wb}

4Ø.  Move Address                                                5
     MOVA{B,W,L=F,Q=D=G,O=H} src.ax, dst.wl

41.  Push Address                                                5
     PUSHA{B,W,L=F,Q=D=G,O=H} src.ax, {-(SP).wl}

42.  Index                                                       1
     INDEX subscript.rl, low.rl, high.rl, size.rl, indexin.rl,
     indexout.wl
```

43. Extract Field 1
    EXTV pos.rl, size.rb, base.vb, {field.rv}, dst.wl

44. Extract Zero-Extended Field 1
    EXTZV pos.rl, size.rb, base.vb, {field.rv}, dst.wl

45. Insert Field 1
    INSV src.rl, pos.rl, size.rb, base.vb, {field.wv}

46. Compare Field 1
    CMPV pos.rl, size.rb, base.vb, {field.rv}, src.rl

47. Compare Zero-Extended Field 1
    CMPZV pos.rl, size.rb, base.vb, {field.rv}, src.rl

48. Find First 2
    FF{S,C} startpos.rl, size.rb, base.vb, {field.rv}, findpos.wl

49. Conditional Branch 12
    B{condition} displ.bb

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

50. Branch With {Byte, Word} Displacement 2
    BR{B,W} displ.bx

51. Jump 1
    JMP dst.ab

52. Branch on Bit 2
    BB{S,C} pos.rl, base.vb, displ.bb, {field.rv}

53. Branch on Bit (and modify without interlock) 4
    BB{S,C}{S,C} pos.rl, base.vb, displ.bb, {field.mv}

54. Branch on Bit (and modify) Interlocked 2
    BB{SS,CC}I pos.rl, base.vb, displ.bb, {field.mv}

```
55.  Branch on Low Bit                                             2
     BLB{S,C} src.rl, displ.bb

56.  Add Compare and Branch                                        7
     ACB{B,W,L,F,D,G,H} limit.rx, add.rx, index.mx, displ.bw
     Compare is LE on positive add, GE on negative
     add.

57.  Add One and Branch Less Than or Equal                         1
     AOBLEQ limit.rl, index.ml, displ.bb

58.  Add One and Branch Less Than                                  1
     AOBLSS limit.rl, index.ml, displ.bb

59.  Subtract One and Branch Greater Than or Equal                 1
     SOBGEQ index.ml, displ.bb

60.  Subtract One and Branch Greater Than                          1
     SOBGTR index.ml, displ.bb

61.  Case                                                          3
     CASE{B,W,L} selector.rx, base.rx, limit.rx, displ.bw-list

62.  Branch to Subroutine With {Byte, Word} Displacement           2
     BSB{B,W} displ.bx, {-(SP).wl}

63.  Jump to Subroutine                                            1
     JSB dst.ab, {-(SP).wl}

64.  Return from Subroutine                                        1
     RSB {(SP)+.rl}

65.  Call Procedure with General Argument List                     1
     CALLG arglist.ab, dst.ab, {-(SP).w*}

66.  Call Procedure with Stack Argument List                       1
     CALLS numarg.rl, dst.ab, {-(SP).w*}

67.  Return from Procedure                                         1
     RET {(SP)+.r*}

68.  Breakpoint Fault                                              1
     BPT {-(KSP).w*}

69.  Halt                                                          1
     HALT {-(KSP).w*}
     Halts in Kernel mode, faults otherwise.
     Assigned opcode 0.

70.  Push Registers                                                1
     PUSHR mask.rw, {-(SP).w*}
```

71. Pop Registers 1
POPR mask.rw, {(SP)+.r*}

72. Move from PSL 1
MOVPSL dst.wl

73. Bit Set PSW 1
BISPSW mask.rw

74. Bit Clear PSW 1
BICPSW mask.rw

75. No Operation 1
NOP

76. Extended Function Call 1
XFC {unspecified operands}

77. Insert Entry in Queue 1
INSQUE entry.ab, pred.ab

78. Insert Entry into Queue at Head, Interlocked 1
INSQHI entry.ab, header.aq

79. Insert Entry into Queue at Tail, Interlocked 1
INSQTI entry.ab, header.aq

8Ø. Remove Entry from Queue 1
REMQUE entry.ab, addr.wl

81. Remove Entry from Queue at Head, Interlocked 1
REMQHI header.aq, addr.wl

82. Remove Entry from Queue at Tail, Interlocked 1
REMQTI header.aq, addr.wl

83. Move Character 3 Operand 1
MOVC3 len.rw, srcaddr.ab, dstaddr.ab, {RØ-5.wl}

84. Move Character 5 operand 1
MOVC5 srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab,
{RØ-5.wl}

85. Move Translated Characters 1
MOVTC srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw,
dstaddr.ab, {RØ-5.wl}

86. Move Translated Until Character 1
MOVTUC srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen,rw,
dstaddr.ab, {RØ-5.wl}

87. Compare Characters 3 Operand 1
CMPC3 len.rw, src1addr.ab, src2addr.ab, {RØ-3.wl}

```
 88.  Compare Characters 5 Operand                                1
      CMPC5 src1len.rw, src1addr.ab, fill.rb, src2len.rw,
      src2addr.ab, {RØ-3.wl}

 89.  Scan Characters                                             1
      SCANC len.rw, addr.ab, tbladdr.ab, mask.rb, {RØ-3.wl}

 9Ø.  Span Characters                                             1
      SPANC len.rw, addr.ab, tbladdr.ab, mask.rb, {RØ-3.wl}

 91.  Locate Character                                            1
      LOCC char.rb, len.rw, addr.ab, {RØ-1.wl}

 92.  Skip Character                                              1
      SKPC char.rb, len.rw, addr.ab, {RØ-1.wl}

 93.  Match Characters                                            1
      MATCHC len1.rw, addr1.ab, len2.rw, addr2.ab, {RØ-3.wl}

 94.  Cyclic Redundancy Check                                     1
      CRC tbl.ab, inicrc.rl, strlen.rw, stream.ab, {RØ-3.wl}

 95.  Move Packed                                                 1
      MOVP len.rw, srcaddr.ab, dstaddr.ab, {RØ-3.wl}

 96.  Compare Packed 3 Operand                                    1
      CMPP3 len.rw, src1addr.ab, src2addr.ab, {RØ-3.wl}

 97.  Compare Packed 4 Operand                                    1
      CMPP4 src1len.rw, src1addr.ab, src2len.rw, src2addr.ab,
      {RØ-3.wl}

 98.  Add Packed 4 Operand                                        1
      ADDP4 addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab, {RØ-3.wl}

 99.  Add Packed 6 Operand                                        1
      ADDP6 add1len.rw, add1addr.ab, add2len.rw, add2addr.ab,
      sumlen.rw, sumaddr.ab, {RØ-5.wl}

1ØØ.  Subtract Packed 4 Operand                                   1
      SUBP4 sublen.rw, subaddr.ab, diflen.rw, difaddr.ab, {RØ-3.wl}

1Ø1.  Subtract Packed 6 Operand                                   1
      SUBP6 sublen.rw, subaddr.ab, minlen.rw, minaddr.ab,
      diflen.rw, difaddr.ab, {RØ-5.wl}

1Ø2.  Multiply Packed                                             1
      MULP mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab,
      prodlen.rw, prodaddr.ab, {RØ-5.wl}

1Ø3.  Divide Packed                                               1
      DIVP divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab,
      quolen.rw, quoaddr.ab, {RØ-5.wl, -16(SP):-1(SP).wb}
```

```
104.  Convert Long to Packed                                    1
      CVTLP src.rl, dstlen.rw, dstaddr.ab, {RØ-3.wl}

105.  Convert Packed to Long                                    1
      CVTPL srclen.rw, srcaddr.ab, {RØ-3.wl}, dst.wl

106.  Convert Packed to Trailing                                2
      Convert Trailing to Packed
      CVT{PT,TP} srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw,
      dstaddr.ab, {RØ-3.wl}

107.  Convert Packed to Leading Separate                        2
      Convert Leading Separate to Packed
      CVT{PS,SP} srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab,
      {RØ-3.wl}

108.  Arithmetic Shift and Round Packed                         1
      ASHP cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw,
      dstaddr.ab, {RØ-3.wl}

109.  Edit Packed to Character String                           1
      EDITPC srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab, {RØ-5.wl}

110.  Probe {Read, Write} Accessability                         2
      PROBE{R,W} mode.rb, len.rw, base.ab

111.  Change Mode                                               4
      CHM{K,E,S,U} param.rw, {-(ySP).w*}
      Illegal on interrupt stack.
      Where y=MINU(x, PSL<current_mode>)

112.  Return from Exception or Interrupt                        1
      REI {(SP)+.r*}

113.  Load Process Context                                      1
      LDPCTX {PCB.r*, -(KSP).w*}
      Legal only on interrupt stack.

114.  Save Process Context
      SVPCTX {(SP)+.r*, PCB.w*}
      Legal only in Kernel mode.

115.  Move To Process Register                                  1
      MTPR src.rl, procreg.rl
      Legal only in Kernel mode.

116.  Move From Processor Register                              1
      MFPR procreg.rl, dst.wl
      Legal only in Kernel mode.
                                                               ---
                                                     Total     3Ø4
```

A.2 OPERAND SPECIFIER NOTATION

The standard VAX notation for operand specifiers is:

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

where:

1. Name is a suggestive name for the operand in the context of the instruction. It is the capitalized name of a register or block for implied operands.

2. Access type is a letter denoting the operand specifier access type.

   a - Calculate the effective address of the specified operand. Address is returned in a pointer which is the actual instruction operand. Context of address calculation is given by data type given by <data type>.
   b - No operand reference. Operand specifier is branch displacement. Size of branch displacement is given by <data type>.
   m - operand is modified (both read and written)
   r - operand is read only
   v - if not "Rn", same as a. If "Rn", R[n+1]'R[n].
   w - operand is written only

3. Data type is a letter denoting the data type of the operand

   b - byte
   d - D_floating
   f - F_floating
   g - G_floating
   h - H_floating
   l - longword
   o - octaword
   q - quadword
   v - field (used only on implied operands)
   w - word
   x - first data type specified by instruction
   y - second data type specified by instruction
   * - multiple longwords (used only on implied operands)

For names, the following names and abbreviations are used:

1. add - addend

2. addr - address

3. arglist - argument list

4. base - base

5. char - character

6. cnt - count

7. dif - difference

8. displ - displacement

9. divd - dividend

10. divr - divisor

11. dst - destination

12. entry - entry

13. esc - escape

14. fill - fill

15. findpos - find position

16. fract - fraction

17. index - index

18. inicrc - initial crc

19. int - integer

20. len - length

21. limit - limit

22. mask - mask

23. min - minuend

24. muld - multiplicand

25. mulr - multiplier

26. mulrx - multiplier extension

27. numarg - number of arguments

28. option - option

29. param - parameter

30. pos - position

31. pred - predecessor

32. procreg - internal processor register

33. prod - product

34. quo - quotient

35. rem - remainder

36. selector - selector

37. size - size

38. src - source

39. startpos - starting position

40. stream - stream

41. strlen - string length

42. sub - subtrahend

43. sum - sum

44. tbl - table

A.3 OPCODE ASSIGNMENTS

SINGLE BYTE OPCODES

Binary	Hex	Mnemonic	Binary	Hex	Mnemonic
00000000	00	HALT	00100000	20	ADDP4
00000001	01	NOP	00100001	21	ADDP6
00000010	02	REI	00100010	22	SUBP4
00000011	03	BPT	00100011	23	SUBP6
00000100	04	RET	00100100	24	CVTPT
00000101	05	RSB	00100101	25	MULP
00000110	06	LDPCTX	00100110	26	CVTTP
00000111	07	SVPCTX	00100111	27	DIVP
00001000	08	CVTPS	00101000	28	MOVC3
00001001	09	CVTSP	00101001	29	CMPC3
00001010	0A	INDEX	00101010	2A	SCANC
00001011	0B	CRC	00101011	2B	SPANC
00001100	0C	PROBER	00101100	2C	MOVC5
00001101	0D	PROBEW	00101101	2D	CMPC5
00001110	0E	INSQUE	00101110	2E	MOVTC
00001111	0F	REMQUE	00101111	2F	MOVTUC
00010000	10	BSBB	00110000	30	BSBW
00010001	11	BRB	00110001	31	BRW
00010010	12	BNEQ,BNEQU	00110010	32	CVTWL
00010011	13	BEQL,BEQLU	00110011	33	CVTWB
00010100	14	BGTR	00110100	34	MOVP
00010101	15	BLEQ	00110101	35	CMPP3
00010110	16	JSB	00110110	36	CVTPL
00010111	17	JMP	00110111	37	CMPP4
00011000	18	BGEQ	00111000	38	EDITPC
00011001	19	BLSS	00111001	39	MATCHC
00011010	1A	BGTRU	00111010	3A	LOCC
00011011	1B	BLEQU	00111011	3B	SKPC
00011100	1C	BVC	00111100	3C	MOVZWL
00011101	1D	BVS	00111101	3D	ACBW
00011110	1E	BGEQU,BCC	00111110	3E	MOVAW
00011111	1F	BLSSU,BCS	00111111	3F	PUSHAW

Binary	Hex	Mnemonic	Binary	Hex	Mnemonic
Ø1ØØØØØØ	4Ø	ADDF2	Ø11ØØØØØ	6Ø	ADDD2
Ø1ØØØØØ1	41	ADDF3	Ø11ØØØØ1	61	ADDD3
Ø1ØØØØ1Ø	42	SUBF2	Ø11ØØØ1Ø	62	SUBD2
Ø1ØØØØ11	43	SUBF3	Ø11ØØØ11	63	SUBD3
Ø1ØØØ1ØØ	44	MULF2	Ø11ØØ1ØØ	64	MULD2
Ø1ØØØ1Ø1	45	MULF3	Ø11ØØ1Ø1	65	MULD3
Ø1ØØØ11Ø	46	DIVF2	Ø11ØØ11Ø	66	DIVD2
Ø1ØØØ111	47	DIVF3	Ø11ØØ111	67	DIVD3
Ø1ØØ1ØØØ	48	CVTFB	Ø11Ø1ØØØ	68	CVTDB
Ø1ØØ1ØØ1	49	CVTFW	Ø11Ø1ØØ1	69	CVTDW
Ø1ØØ1Ø1Ø	4A	CVTFL	Ø11Ø1Ø1Ø	6A	CVTDL
Ø1ØØ1Ø11	4B	CVTRFL	Ø11Ø1Ø11	6B	CVTRDL
Ø1ØØ11ØØ	4C	CVTBF	Ø11Ø11ØØ	6C	CVTBD
Ø1ØØ11Ø1	4D	CVTWF	Ø11Ø11Ø1	6D	CVTWD
Ø1ØØ111Ø	4E	CVTLF	Ø11Ø111Ø	6E	CVTLD
Ø1ØØ1111	4F	ACBF	Ø11Ø1111	6F	ACBD
Ø1Ø1ØØØØ	5Ø	MOVF	Ø111ØØØØ	7Ø	MOVD
Ø1Ø1ØØØ1	51	CMPF	Ø111ØØØ1	71	CMPD
Ø1Ø1ØØ1Ø	52	MNEGF	Ø111ØØ1Ø	72	MNEGD
Ø1Ø1ØØ11	53	TSTF	Ø111ØØ11	73	TSTD
Ø1Ø1Ø1ØØ	54	EMODF	Ø111Ø1ØØ	74	EMODD
Ø1Ø1Ø1Ø1	55	POLYF	Ø111Ø1Ø1	75	POLYD
Ø1Ø1Ø11Ø	56	CVTFD	Ø111Ø11Ø	76	CVTDF
Ø1Ø1Ø111	57	RESERVED to DEC	Ø111Ø111	77	RESERVED to DEC
Ø1Ø11ØØØ	58	ADAWI	Ø1111ØØØ	78	ASHL
Ø1Ø11ØØ1	59	RESERVED to DEC	Ø1111ØØ1	79	ASHQ
Ø1Ø11Ø1Ø	5A	RESERVED to DEC	Ø1111Ø1Ø	7A	EMUL
Ø1Ø11Ø11	5B	RESERVED to DEC	Ø1111Ø11	7B	EDIV
Ø1Ø111ØØ	5C	INSQHI	Ø11111ØØ	7C	CLRQ,CLRD,CLRG
Ø1Ø111Ø1	5D	INSQTI	Ø11111Ø1	7D	MOVQ
Ø1Ø1111Ø	5E	REMQHI	Ø111111Ø	7E	MOVAQ,MOVAD,MOVAG
Ø1Ø11111	5F	REMQTI	Ø1111111	7F	PUSHAQ,PUSHAD,PUSHAG

Binary	Hex	Mnemonic	Binary	Hex	Mnemonic
10000000	80	ADDB2	10100000	A0	ADDW2
10000001	81	ADDB3	10100001	A1	ADDW3
10000010	82	SUBB2	10100010	A2	SUBW2
10000011	83	SUBB3	10100011	A3	SUBW3
10000100	84	MULB2	10100100	A4	MULW2
10000101	85	MULB3	10100101	A5	MULW3
10000110	86	DIVB2	10100110	A6	DIVW2
10000111	87	DIVB3	10100111	A7	DIVW3
10001000	88	BISB2	10101000	A8	BISW2
10001001	89	BISB3	10101001	A9	BISW3
10001010	8A	BICB2	10101010	AA	BICW2
10001011	8B	BICB3	10101011	AB	BICW3
10001100	8C	XORB2	10101100	AC	XORW2
10001101	8D	XORB3	10101101	AD	XORW3
10001110	8E	MNEGB	10101110	AE	MNEGW
10001111	8F	CASEB	10101111	AF	CASEW
10010000	90	MOVB	10110000	B0	MOVW
10010001	91	CMPB	10110001	B1	CMPW
10010010	92	MCOMB	10110010	B2	MCOMW
10010011	93	BITB	10110011	B3	BITW
10010100	94	CLRB	10110100	B4	CLRW
10010101	95	TSTB	10110101	B5	TSTW
10010110	96	INCB	10110110	B6	INCW
10010111	97	DECB	10110111	B7	DECW
10011000	98	CVTBL	10111000	B8	BISPSW
10011001	99	CVTBW	10111001	B9	BICPSW
10011010	9A	MOVZBL	10111010	BA	POPR
10011011	9B	MOVZBW	10111011	BB	PUSHR
10011100	9C	ROTL	10111100	BC	CHMK
10011101	9D	ACBB	10111101	BD	CHME
10011110	9E	MOVAB	10111110	BE	CHMS
10011111	9F	PUSHAB	10111111	BF	CHMU

Binary	Hex	Mnemonic
11000000	C0	ADDL2
11000001	C1	ADDL3
11000010	C2	SUBL2
11000011	C3	SUBL3
11000100	C4	MULL2
11000101	C5	MULL3
11000110	C6	DIVL2
11000111	C7	DIVL3
11001000	C8	BISL2
11001001	C9	BISL3
11001010	CA	BICL2
11001011	CB	BICL3
11001100	CC	XORL2
11001101	CD	XORL3
11001110	CE	MNEGL
11001111	CF	CASEL
11010000	D0	MOVL
11010001	D1	CMPL
11010010	D2	MCOML
11010011	D3	BITL
11010100	D4	CLRL,CLRF
11010101	D5	TSTL
11010110	D6	INCL
11010111	D7	DECL
11011000	D8	ADWC
11011001	D9	SBWC
11011010	DA	MTPR
11011011	DB	MFPR
11011100	DC	MOVPSL
11011101	DD	PUSHL
11011110	DE	MOVAL,MOVAF
11011111	DF	PUSHAL,PUSHAF

Binary	Hex	Mnemonic
11100000	E0	BBS
11100001	E1	BBC
11100010	E2	BBSS
11100011	E3	BBCS
11100100	E4	BBSC
11100101	E5	BBCC
11100110	E6	BBSSI
11100111	E7	BBCCI
11101000	E8	BLBS
11101001	E9	BLBC
11101010	EA	FFS
11101011	EB	FFC
11101100	EC	CMPV
11101101	ED	CMPZV
11101110	EE	EXTV
11101111	EF	EXTZV
11110000	F0	INSV
11110001	F1	ACBL
11110010	F2	AOBLSS
11110011	F3	AOBLEQ
11110100	F4	SOBGEQ
11110101	F5	SOBGTR
11110110	F6	CVTLB
11110111	F7	CVTLW
11111000	F8	ASHP
11111001	F9	CVTLP
11111010	FA	CALLG
11111011	FB	CALLS
11111100	FC	XFC
11111101	FD	ESCD to DEC
11111110	FE	ESCE to DEC
11111111	FF	ESCF to DEC

TWO BYTE OPCODES

Hex	Mnemonic	Hex	Mnemonic
ØØFD to 31FD	RESERVED to DIGITAL		
32FD	CVTDH	33FD	CVTGF
34FD to 3FFD	RESERVED to DEC		
4ØFD	ADDG2	6ØFD	ADDH2
41FD	ADDG3	61FD	ADDH3
42FD	SUBG2	62FD	SUBH2
43FD	SUBG3	63FD	SUBH3
44FD	MULG2	64FD	MULH2
45FD	MULG3	65FD	MULH3
46FD	DIVG2	66FD	DIVH2
47FD	DIVG3	67FD	DIVH3
48FD	CVTGB	68FD	CVTHB
49FD	CVTGW	69FD	CVTHW
4AFD	CVTGL	6AFD	CVTHL
4BFD	CVTRGL	6BFD	CVTRHL
4CFD	CVTBG	6CFD	CVTBH
4DFD	CVTWG	6DFD	CVTWH
4EFD	CVTLG	6EFD	CVTLH
4FFD	ACBG	6FFD	ACBH

5ØFD	MOVG	7ØFD	MOVH
51FD	CMPG	71FD	CMPH
52FD	MNEGG	72FD	MNEGH
53FD	TSTG	73FD	TSTH
54FD	EMODG	74FD	EMODH
55FD	POLYG	75FD	POLYH
56FD	CVTGH	76FD	CVTHG
57FD	RESERVED to DEC	77FD	RESERVED to DEC
58FD	RESERVED to DEC	78FD	RESERVED to DEC
59FD	RESERVED to DEC	79FD	RESERVED to DEC
5AFD	RESERVED to DEC	7AFD	RESERVED to DEC
5BFD	RESERVED to DEC	7BFD	RESERVED to DEC
5CFD	RESERVED to DEC	7CFD	CLRH,CLRO
5DFD	RESERVED to DEC	7DFD	MOVO
5EFD	RESERVED to DEC	7EFD	MOVAH,MOVAO
5FFD	RESERVED to DEC	7FFD	PUSHAH,PUSHAO
8ØFD to 97FD	RESERVED to DIGITAL		
98FD	CVTFH	99FD	CVTFG
9AFD to F5FD	RESERVED to DIGITAL		
F6FD	CVTHF	F7FD	CVTHD
F8FD to FCFF	RESERVED to DIGITAL		
FDFF	BUGL (used by VMS for BUGCHECK)	FEFF	BUGW
FFFF	RESERVED for all time		

A.4 INSTRUCTIONS USABLE TO REFERENCE I/O SPACE

Some of the instructions are not usable to reference I/O space. The reasons for this are:

1. String instructions are restartable via PSL<FPD>

2. The instruction is not in the kernel set

3. The PC, SP, or PCBB can not point to I/O space

4. I/O space does not support operand types of quad, floating, field, or queue; nor can the position, size, length, or base of them be from I/O space

5. The instruction may be interruptible because it is potentially a slow instruction in some implementations

6. Only instructions with a maximum of one modify or write destination can be used. The destination must be the last operand

For any memory reference to I/O space, the programmer must use an instruction from the following lists and must ensure that no interrupts or faults will occur, including page faults, after the first I/O space reference. To ensure no interrupts, the programmer must avoid operand specifier modes 9, 11, 13, and 15, and these modes indexed. (Symbolically, these are @(Rn)+, @B^D(Rn), @W^D(Rn), and @L^D(Rn), and these indexed.) The hardware may allow interrupts for these modes in order to minimize interrupt latency. For the instructions in the following lists, the hardware ensures that no other interrupts will occur after the first I/O space access.

Since these instructions are not interruptable after I/O space accesses (except for the addressing modes above), their execution will extend the interrupt latency. The programmer should make some effort to keep them short by minimizing the number of memory references. Use RØ through R13 instead, for example.

Instructions for which any explicit operand can be in I/O space:

MOV{B,W,L}, PUSHL, CLR{B,W,L}, MNEG{B,W,L}, MCOM{B,W,L}, MOVZ{BW,BL,WL}, CVT{BW,BL,WB,WL,LB,LW}, CMP{B,W,L}, TST{B,W,L}, ADD{B,W,L}2, ADD{B,W,L}3, ADAWI, INC{B,W,L}, ADWC, SUB{B,W,L}2, SUB{B,W,L}3, DEC{B,W,L}, SBWC, BIT{B,W,L}, BIS{B,W,L}2, BIS{B,W,L}3, BIC{B,W,L}2, BIC{B,W,L}3, XOR{B,W,L}2, XOR{B,W,L}3, MOVA{B,W,L}, MOVAQ, PUSHA{B,W,L}, PUSHAQ, CASE{B,W,L}, MOVPSL, BISPSW, BICPSW, CHM{K,E,S,U} PROBE{R,W}, MTPR, MFPR

Instructions for which all operands except the branch displacement can be in I/O space:

BLB{S,C}

Instruction for which some operand can be in I/O space:

```
        XFC     (depending on implementation)
        REMQUE  addr (destination)
        REMQHI  addr (destination)
        REMQTI  addr (destination)
```

Notwithstanding the above rules, it is possible for a specific hardware implementation to execute macro code from the I/O space and/or to allow the stack or PCB to be in I/O space. This might, for example, be used as part of the bootstrap process. If this is done, then it is valid for software to transfer to this code.

INDEX

VAX-11 ARCHITECTURE REFERENCE MANUAL
REVISION 6.1

READER'S COMMENTS

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our manuals.

What is your general reaction to this manual? (format, accuracy, completeness, organization, etc.) ______

What features are most useful? ______

Does the publication satisfy your needs? ______

What errors have you found? ______

Additional comments ______

Name ______
Title ______
Company ______ Dept. ______
Address ______
City ______ State ______ Zip ______

Do Not Tear - Fold Here and Staple

Fold Here

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD, MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
VAX ARCHITECTURE MANAGEMENT
1925 ANDOVER STREET
TW/B05
TEWKSBURY, MA 01876

Digital Equipment Corporation • Bedford, MA 01730