VAX - 11

SYSTEM REFERENCE MANUAL

19 FEB 1979

Revision 5

Digital Equipment Corporation, Maynard, Massachusetts

Revision 1, Sept, 1975
Revision 3, June, 1976
Revision 4, May, 1977
Revision 5, Feb, 1979

PREFACE

The VAX-11 is a family of upward-compatible computer systems. It is a natural outgrowth of and is heavily 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-11/780 Technical Summary. It contains a 30 page summary of this manual which is suitable for marketing and sales use, but is not sufficiently detailed for software or hardware implementation.

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 (not included in this release) contains the Glossary of all terms used in VAX-11. Appendix B details the assembler notation sufficiently for a full understanding of the examples in the manual. Appendix C includes the software calling sequence standard. Appendix D documents the condition handling facility. Appendix E details the

rules for subsetting instructions in various implementations of the VAX-11 family. Appendix F is a summary of the instructions, their operands, and the encoding. It is suitable to be used to construct an "instruction card". Appendix G (not included in this release) documents the notation used in the formal instruction descriptions. Appendix H includes examples of how to use the VAX-11 instruction set to build multi-precision integer arithmetic. Appendix I is a conversion guide for converting PDP-11 machine language programs to VAX-11. Appendix J is the rules which the operating system must obey to use the memory management system without introducing protection holes. Appendix K gives programming examples.

In addition to being a manual for programmers, this manual is also the architectural control document for all machines built in the VAX-11 family. It delineates the permitted variations between processor implementations. These decisions were reached taking into account tradeoffs between software development costs and hardware manufacturing costs over a wide range of technologies.

In using the System Reference Manual as a reference, there is a possible ambiguity because a single specification may occur several times. Although every effort has been made to ensure that the specifications are consistent, conflicting specification is possible. In order to resolve any conflicts, the following precedence order is established:

1. Appendix F

2. Formal description in chapters 1-11 in that order

3. English description in chapters 1-11 in that order

4. Appendices A-E and G-K in that order

Any errors, inconsistencies, or ambiguitities should be brought to the attention of:

Ted Taylor
TW/A08

Title: VAX-11 System Reference Manual -- Rev 5

Specification Status:

Architectural Status: under ECO control

File: SRPRR5.RNO

PDM #: not used

Date: 19-Feb-79

Superseded Specs: SRM Rev 4

Abstract: This document is revision 5 of the definition of the Hardware Architecture for the VAX-11 computer family. It is composed on a set of individual architectural specifications (chapters) which are published as an entity known as the System Reference Manual. Refer to the Preface for a description of the manual as a whole or to the abstracts of the individual specifications to understand the purpose and content of each chapter.

Revision History:

Rev.#	Description	Author	Revised Date
Rev 1	Original publication	VAXA	Oct-75
Rev 2	Internal, for review	VAXA	Mar-76
Rev 3	Result of implementor's review	April T.F.	13-Jun-76
Rev 4	ECO's	VAXA	12-May-77
Rev 5	ECO's, G & H,interlocked queue	Bhandarkar	19-Feb-79

+---------------+
| d i g i t a l |
+---------------+

interoffice memorandum

To: VAX SRM Distribution

Date: 12 February 79
From: Dileep Bhandarkar
Dept: VAX/PDP-11 Sys. Arch.
DTN: 247-2021
Loc/Mail Stop: TW/A08

Subj: VAX SRM

This package contains updated versions of Chapters 1, 2, 3, 4, and 6, Appendices C, E, and F, and a new Index and Contents. The SRM now contains specifications for instructions and features not available on machines currently being shipped. Please continue to treat the SRM as a Company Confidential document.

CONTENTS

Title: VAX-11 Introduction - Rev 5

Specification Status: Fully approved

Architectural Status: under ECO control

File: SR1R5.RNO

PDM #: not used

Date: 28-Oct-78

Superseded Specs: Rev 4

Author: W. Strecker

Typist: Betty Call

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 1 gives the goals which guided the design of the VAX-11 architecture. It then defines the terminology which is key to understanding the remainder of the System Reference Manual.

Revision History:

Rev.#	Description	Author	Revised Date
Rev 1	Distributed	Strecker	25-Sep-75
Rev 2	ECOs 1-11	Strecker	1-Mar-76
Rev 3	ECOs 12-18 plus April Meeting	Strecker	10-Jun-76
Rev 4		Strecker	28-Feb-77
Rev 5	Editorial	Bhandarkar	28-Oct-78

Rev 4 to Rev 5:

1. UNDEFINED must not hang
2. MBZ checks mandatory for non-kernel mode fields
3. MBZ checks optional for kernel mode fields

Rev 3 to Rev 4:

Rev 2 to Rev 3:

1. Remove 1000:1 goal and multiprocessors
2. Change UNDEFINED result to UNPREDICTABLE
3. Define RESERVED

Rev 1 to Rev 2

1. Define MBZ

[End of SR1R5.RNO]

CHAPTER 1

INTRODUCTION

28-Oct-78 -- Rev 5

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

1.2 TERMINOLOGY AND CONVENTIONS

A general glossary of VAX-11 terminology appears in Appendix A. However, several important terms and conventions are outlined here.

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.

> \A note on the manual format: At certain points in the manual comments on why certain decisions were made, unresolved issues, etc., are included between a pair of back slants.\

[End of Chapter 1]

Title: VAX-11 Basic Architecture -- Rev 5

Specification Status:

Architectural Status: under ECO control

File: SR2R5.RNO

PDM #: not used

Date: 31-Jan-79

Superseded Specs: Rev 4

Author: W. Strecker

Typist: B. Call

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 2 specifies the formats of each of the data types supported by the VAX-11 architecture. It also defines the processor state which includes the general registers and the processor status word. It briefly describes the instruction formats, interrupts and the I/O structure.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Distributed	Strecker	25-Sep-75
Rev 2	ECOs 1-11	Strecker	9-Mar-76
Rev 3	ECOs 12-18 and April Meeting	Strecker	3-Jun-76
Rev 4	ECOs	Strecker	5-Apr-77
Rev 5	Extended range floating point	Bhandarkar	31-Jan-79

Rev 4 to Rev 5

1. Rename floating and double to F_floating and D_floating
2. Add G_floating data type
3. Add H_floating data type
4. Add FF bit to PSW
5. ECO to Overpunch alternate character set

Rev 3 to Rev 4:

1. Replace zoned with packed (decimal data ECO).
2. Change CF to FP (CF ECO).
3. Add definitions of overpunch and zoned formats. Include alternate forms of overpunch.

Rev 2 to Rev 3:

1. Remove AL field, pointer.
2. Numeric from left separate to right zoned.
3. Remove DZ,FV.
4. IV and DV conditionally enabled by Call instructions.
5. Remove Round from C.
6. All overflowing instructions trap.
7. Remove LP.
8. Expand to 32 interrupt levels.
9. Zero length permitted for numeric and pocked decimal strings.
10. Zero length field causes 0 memory references.

Rev 1 to Rev 2:

1. Clarify AL field, add pointer.
2. Add Quad.
3. Clarify floating range.
4. Don't wrap field.

5. Define nibble.

6. Combine ND,FD,ID into DZ.

7. Remove TBR.

8. C gets Round bit.

9. Registers look like memory.

10. All instructions can do I/O.

[End of SR2R5.RNO]

CHAPTER 2

BASIC ARCHITECTURE

31-Jan-79 -- Rev 5

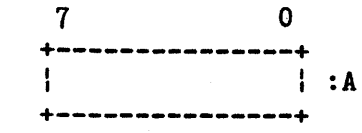
2.1 ADDRESSING

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

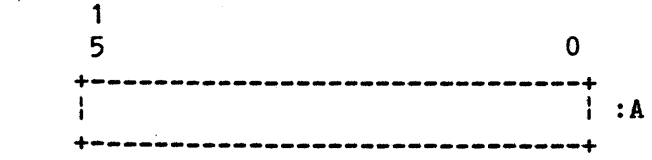
A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from the right 0 through 7:



A byte is specified by its address A. When interpreted arithmetically, a byte is a twos complement integer with bits of increasing significance going 0 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 0 through 7. The value of the unsigned integer is in the range 0 through 255.

2.2.2 Word

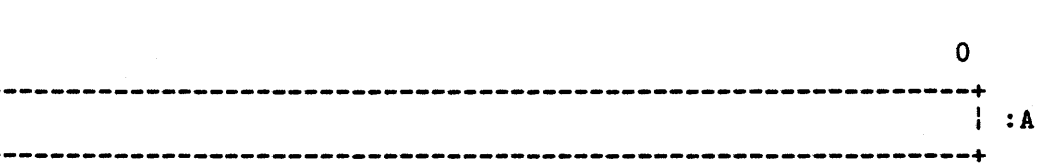
A word is 2 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 15:



A word is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a word is a twos complement integer with bits of increasing significance going 0 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 0 through 15. The value of the unsigned integer is in the range 0 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 0 through 31:

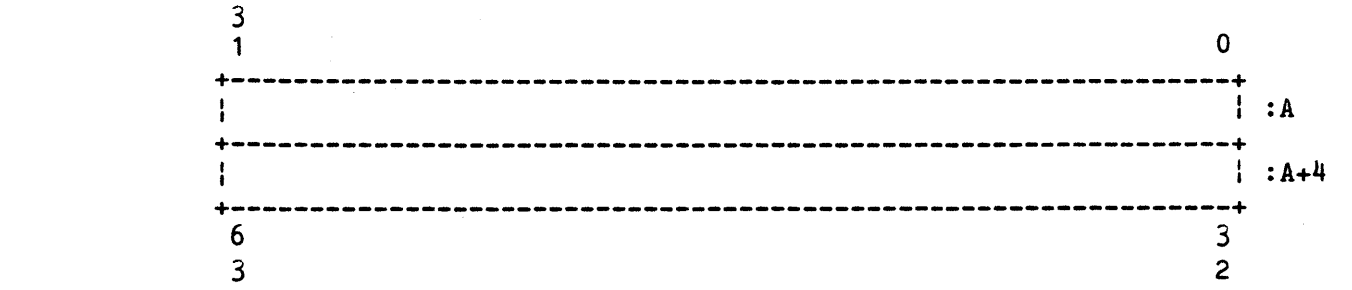


A longword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a longword is a twos complement integer with bits of increasing significance going 0 through 30 and bit 31 the sign bit. The value of the integer is in the range -2,147,483,648 through 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 0 through 31. The value of the unsigned integer is in the range 0 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 0 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 0 through 63:

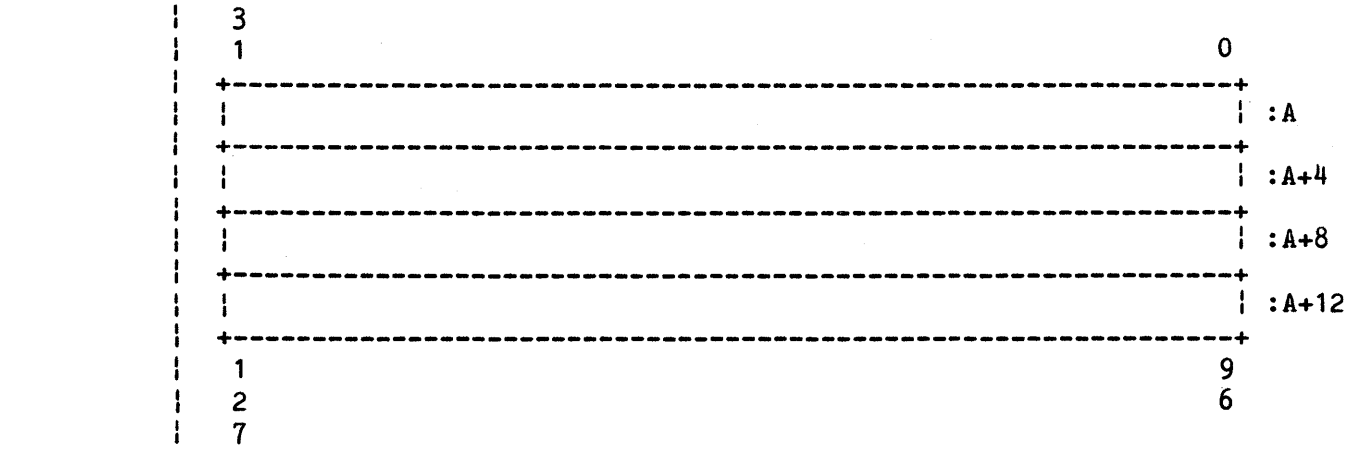


A quadword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a quadword is a twos complement integer with bits of increasing significance going 0 through 62 and bit 63 the sign bit. The value of the integer is in the range -2**63 to 2**63-1. The quadword data type is not fully supported by VAX-11 instructions.

Largest unsigned number = 18446744073709551615 = $2^{64}-1$

2.2.5 Octaword

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

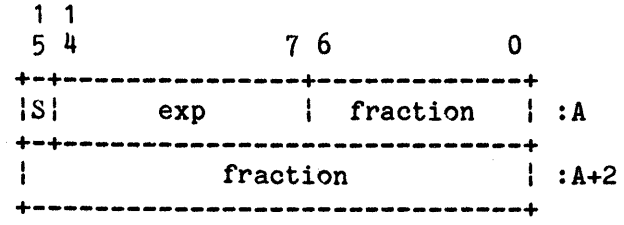


A octaword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a octaword is a twos complement integer with bits of increasing significance going 0 through 126 and bit 127 the sign bit. The value of the integer is in the range -2**127 to 2**127-1. The octaword data type is not fully supported by VAX-11 instructions.

$2^{127}-1$ = 340282366920938463463374607431768211455

2.2.6 F_floating

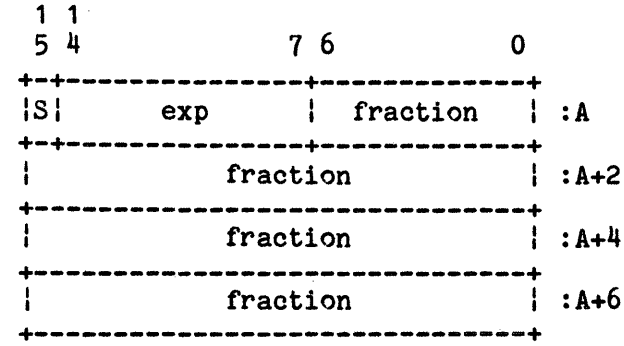
A F_floating datum is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 31.



A F_floating datum is specified by its address A, the address of the byte containing bit 0. 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 6:0 and 31:16 a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 16 through 31 and 0 through 6. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0 together with a sign bit of 0, is taken to indicate that the F_floating datum has a value of 0. Exponent values of 1 through 255 indicate true binary exponents of -127 through +127. 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 F_floating datum is in the approximate range .29*10**-38 through 1.7*10**38. The precision of a F_floating datum is approximately one part in 2**23, i.e., typically 7 decimal digits.

2.2.7 Double Floating (D_floating)

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

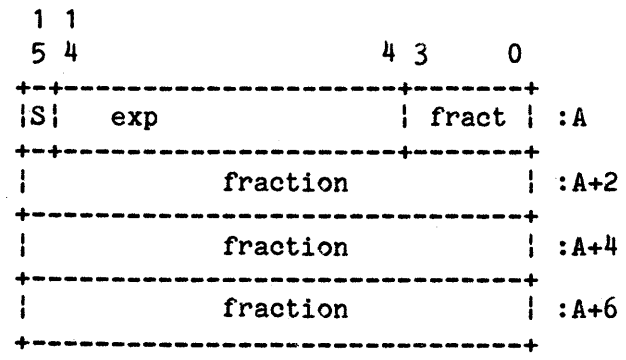


A D_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a D_floating datum is identical to a floating datum except for an additional 32 low significance fraction bits. Within the fraction, bits of increasing significance go 48

through 63, 32 through 47, 16 through 31, and 0 through 6. The exponent conventions, and approximate range of values is the same for D_floating as F_floating. The precision of a D_floating datum is approximately one part in 2**55, i.e., typically 16 decimal digits.

2.2.8 G_floating

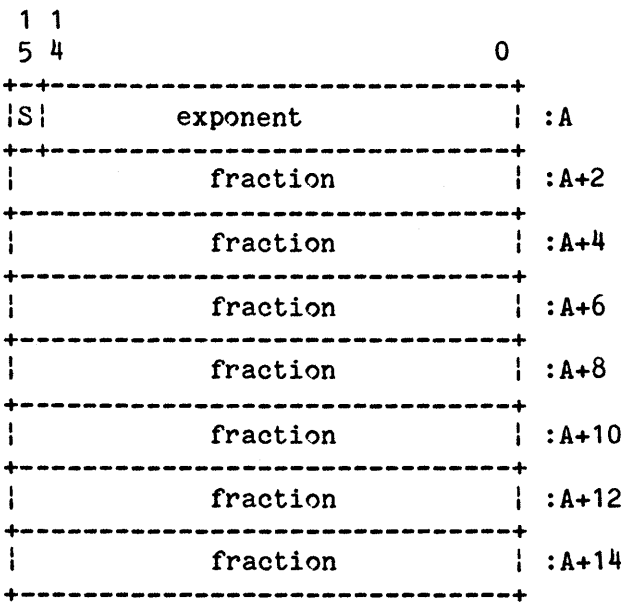
A G_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 63:



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:

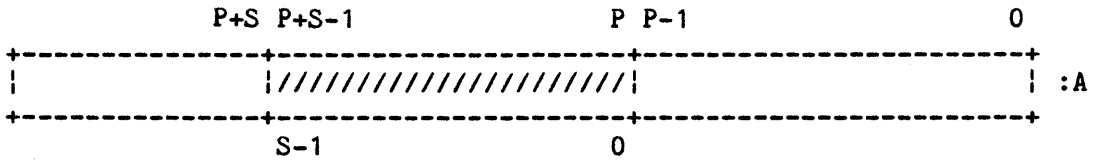


A H_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a H_floating datum is sign magnitude with bit 15 the sign bit, bits 14:0 an excess 16384 binary exponent, and bits 127:16 a normalized 113-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go 112 through 127, 96 through 111, 80 through 95, 64 through 79,48 through 63, 32 through 47, and 16 through 31. The 15-bit exponent field encodes the values 0 through 32767. An exponent value of 0 together with a sign bit of 0, is taken to indicate that the H_floating datum has a value of 0. Exponent values of 1 through 32767 indicate true binary exponents of -16383 through +16383. 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 H_floating datum is in the approximate range .84*10**-4932 through .59*10**4932. The precision of a H_floating datum is approximately one part in 2**112, i.e., typically 33 decimal digits.

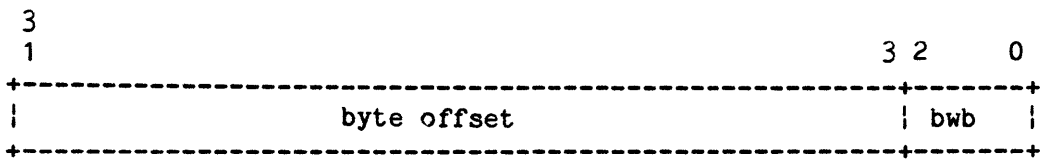
.5948 6574 7678 6158 8254 2879 6633 1400 35
.8405 2578 5778 0232 7656 5669 4543 3

2.2.10 Variable Length Bit Field

A variable bit field is 0 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 0 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.



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:

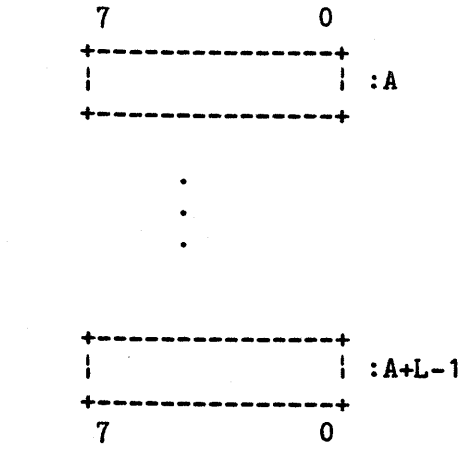


The sign extended 29-bit byte offset is added to the address A and the resulting address specifies the byte in which the field begins. The 3-bit bit-within-byte field encodes the starting position (0 through 7) of the field within that byte. The VAX-11 field instructions provide direct support for the 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 0 through S-2; bit S-1 is the sign bit. When interpreted as an unsigned integer, bits of increasing significance go from 0 to S-1. A field of size 0 has a value identically equal to 0.

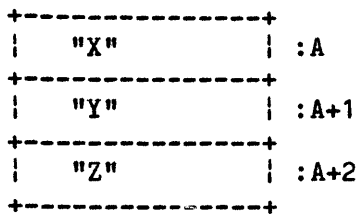
A variable bit field may be contained in 1 to 5 bytes. From a memory management point of view (Chapter 5) only the minimum number of bytes necessary to contain the field is actually referenced.

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:



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



The length L of a string is in the range 0 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.

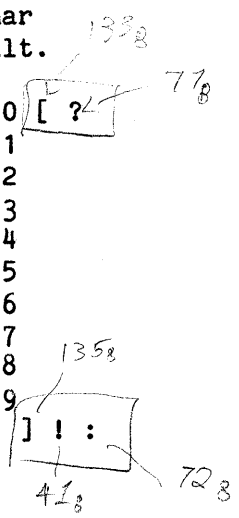
All bytes of a trailing numeric string, except the least significant digit byte, must contain an ASCII decimal digit character (0-9). The representation for the high order digits is:

digit	decimal	hex	ASCII character
0	48	30	0
1	49	31	1
2	50	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

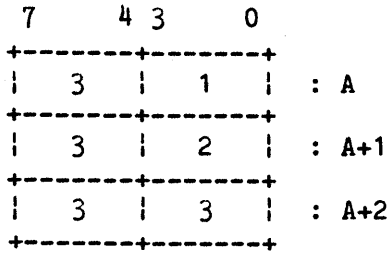
digit	Zoned Numeric Format: decimal	hex	ASCII char	Overpunch Format: decimal	hex	ASCII norm	ASCII char alt.
0	48	30	0	123	7B	{	0
1	49	31	1	65	41	A	1
2	50	32	2	66	42	B	2
3	51	33	3	67	43	C	3
4	52	34	4	68	44	D	4
5	53	35	5	69	45	E	5
6	54	36	6	70	46	F	6
7	55	37	7	71	47	G	7
8	56	38	8	72	48	H	8
9	57	39	9	73	49	I	9
-0	112	70	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	80	50	P	
-8	120	78	x	81	51	Q	
-9	121	79	y	82	52	R	



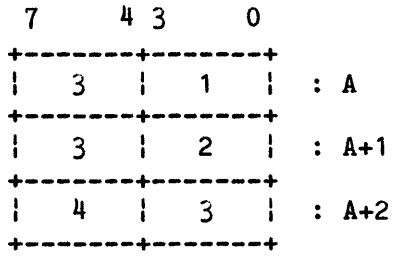
The length L of a trailing numeric string must be in the range 0 to 31 (0 to 31 digits). The value of a 0 length string is identically 0.

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

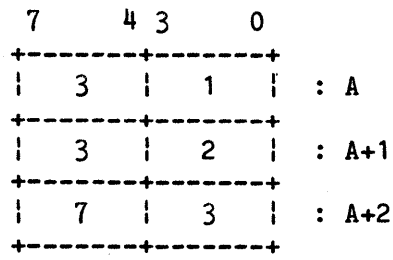


Overpunch Format

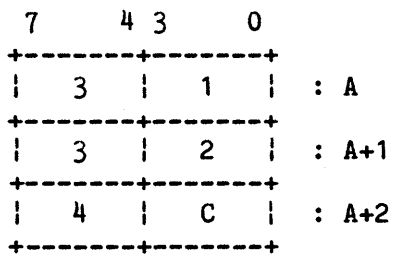


and "-123" is represented :

Zoned Format



Overpunch Format



2.2.12.1 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:

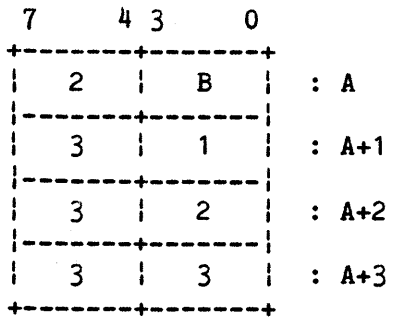
Sign	decimal	hex	ASCII character
+	43	2B	+
+	32	20	<blank>
-	45	2D	-

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

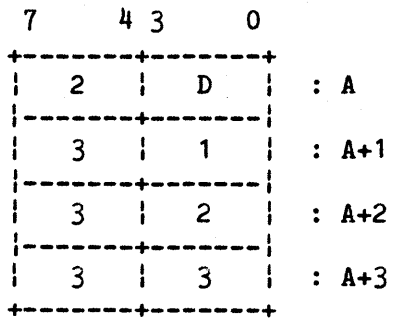
digit	decimal	hex	ASCII character
0	48	30	0
1	49	31	1
2	50	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The length L of a leading separate numeric string must be in the range 0 to 31 (0 to 31 digits). The value of a 0 length string is identically 0.

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:



and "-123" is:



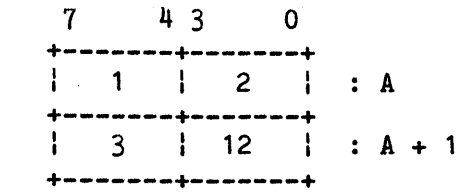
2.2.13 Packed Decimal String

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 3:0) of the last (highest addressed) byte which must contain a sign. The representation for the digits and sign is:

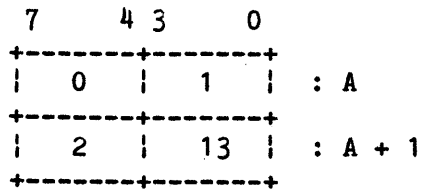
digit or sign	decimal	hex
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
+	10,12,14 or 15	A,C,E, or F
-	11 or 13	B, or D

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 0 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 "0" 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:



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

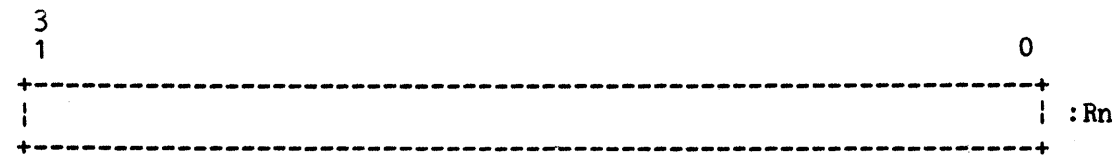

7
4 3
0
0
1
: A
2
13
: A + 1

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:



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 Appendix C) 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].

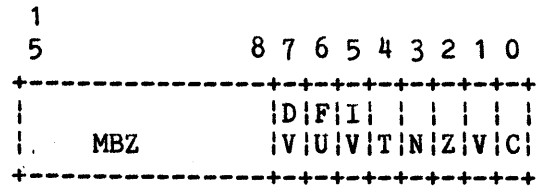
When an octaword or a H_floating datum is stored in register R[n], it is actually stored in adjacent registers R[n], R[n+1], R[n+2], and R[n+3]. Because of restrictions on the specification of PC (see Chapter 3) wraparound from PC to R0 is UNPREDICTABLE. Bits 31:0 of the datum are stored in bits 31:0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



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, the T (trace) bit forces a trace trap at the end of each instruction execution. When clear, no trap occurs. See Chapter 6 for additional information on the trace trap.

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 exception if the result of a floating point instruction is too small in magnitude to be represented in the result operand. On the original VAX-11/780's (not implementing the revised floating point architecture), a trap after the execution of the instruction that produced an underflowed result occurs. On all other VAX processors, a fault occurs. When FU is clear, no trap or fault occurs.

\The floating point architecture was revised in January 1979 to take faults instead of traps on floating exceptions for the original data types (F_floating and D_floating) as well as the two new data types (G_floating and H_floating). Modified versions of the VAX-11/780 with the revised architecture might be built.\

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. On the original VAX-11/780, a trap is also forced after the execution of a floating division instruction which has a zero divisor. On all other VAX processors, a fault occurs on a floating division instruction which has a zero divisor.

2.5.2 Floating Overflow

A floating overflow trap (original VAX-11/780) or fault (all other VAX processors) 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 0 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 F 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.

[End of Chapter 2]

Title: VAX-11 Instruction Formats and Addressing Modes -- Rev 6

Specification Status: Fully approved

Architectural Status: under ECO control

File: SR3R6.RNO

PDM #: not used

Date: 14-Jul-78

Superseded Specs: Rev 5

Author: W. Strecker

Typist: Betty Call

Reviewer(s): R. Brender, P. Conklin, D. Cutler, R. Grove, D. Hustvedt, M. Jack, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, W. Strecker

Abstract: Chapter 3 gives the formats of opcodes and operand specifiers. It describes the types of operand specifiers. It describes the behavior of each addressing mode in both text and a formal notation. It lists a set of conventions applied to operand specifier evaluation.

Revision History:

Rev.#	Description	Author	Revised Date
Rev 1	Distributed	Strecker	25-Sep-75
Rev 2	ECOs 1-11	Strecker	10-Mar-76
Rev 3	ECOs 12-18, April Meeting	Strecker	12-May-76
Rev 4	Address Mode Changes	Strecker	3-Jun-76
Rev 5	Editorial	Strecker	7-Feb-77
Rev 6	128 bit floating data type ECO	Bhandarkar	14-Jul-78

Rev 5 to Rev 6:

1. Add G_floating and H_floating
2. Add octaword

Rev 4 to Rev 5:

1. Editorial change to Summary of Mode Addressing

Rev 3 to Rev 4:

1. Change floating literals
2. Remove argument mode, local mode and post-index mode
3. Add byte, word, longword displacement deferred mode
4. Add autoincrement deferred mode
5. Write of immediate UNPREDICATBLE
6. Register deferred of PC UNPREDICTABLE
7. PC index register gives reserved operand fault (future escape)
8. SP index register OK
9. Same register for base and index OK for modes other than autoincrement, autodecrement, and autoincrement deferred.
10. Write to PC of operand taking 2 registers result in R0 UNPREDICTABLE.

Rev 2 to Rev 3:

1. Note that modify is not under memory interlock
2. Change pointer to longword, remove <27:0> and EAL
3. Change R'R+1 to R[n+1]'Rn (typo).
4. Clarify what is UNPREDICTABLE
5. Modify and write of immediate is UNPREDICTABLE
6. Change hex modes to decimal; add table
7. Change R to LP in local mode (typo)
8. Clarify value of PC in displacement and branch displacement

9. Change address mode abort to address mode fault

Rev 1 to Rev 2:

1. Remove R-R mode
2. Reduce local 6 bits to 5
3. Add (R)
4. Add E[Rx]
5. Rename base-indexed to post-indexed
6. Eliminate base displacement. (record indexing)
7. Add 28-bit address arithmetic
8. Make branches 8 bit and 16 bit displacements

[End of SR3R6.RNO]

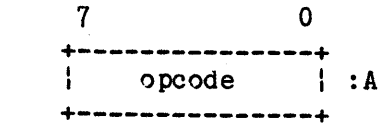
CHAPTER 3

INSTRUCTION FORMATS AND ADDRESSING MODES

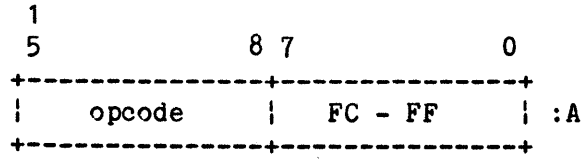
14-Jul-78 -- Rev 6

3.1 OPCODE FORMATS

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



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:



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. Branch - no operand is accessed. The operand specifier itself is a branch displacement.

Types 1 - 4 are termed general mode addressing and are discussed in Section 3.4. Type 5 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.

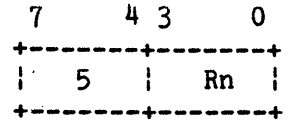
(x)	- the contents of a location in memory whose address is x.
{ }	- arithmetic parentheses used to indicate precedence
SEXT(x)	- x is sign extended to size of operand needed
ZEXT(x)	- x is zero extended to size of operand needed
OA	- operand address
!	- comment delimiter

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:



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):

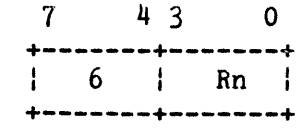
```
operand = Rn                          !if one register
          or
          R[n+1]'Rn                   !if two registers
          or
          R[n+3]'R[n+2]'R[n+1]'Rn     !if four registers
```

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 R0 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 R0, 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 R0 are UNPREDICTABLE; and, if SP is used in register mode for a write access type operand which takes 4 adjacent registers, the contents of R0 and R1 are UNPREDICTABLE.

The assembler notation (See Appendix B) for register mode is Rn.

3.4.2 Register Deferred Mode

The operand specifier format is:



No specifier extension follows.

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

OA = Rn

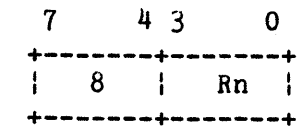
operand = (OA)

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 (See Appendix B) for register deferred mode is (Rn).

3.4.3 Autoincrement Mode

The operand specifier format is:



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, D_floating and G_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)

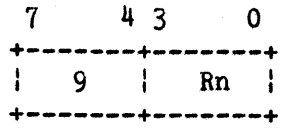
Immediate mode may not be used for operands of modify or write access type. If immediate mode is used for an operand of modify access type,

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 (See Appendix B) 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:



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

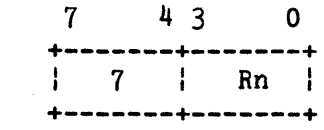
In autoincrement deferred mode addressing, the address of the operand is the contents of a longword whose address is the contents of register n. After the operand address is determined, 4 (the size in bytes of a longword address) is added to the contents of register n and the contents of register n is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

```
OA = (Rn)

Rn <- Rn + 4

operand = (OA)
```

The assembler notation (See Appendix B) 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:



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:

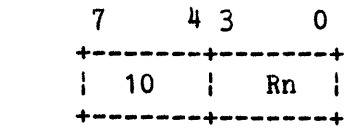
```
Rn  <- Rn  - size

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.

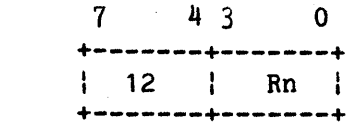
The assembler notation (See Appendix B) for autodecrement mode is -(Rn).

3.4.6 Displacement Mode

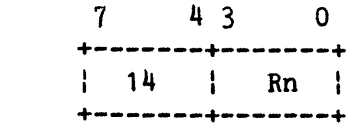
There are 3 operand specifier formats:



The specifier extension is a signed byte displacement. This is termed byte displacement mode.



The specifier extension is a signed word displacement. This is termed word displacement mode.



The specifier extension is a longword displacement. 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:

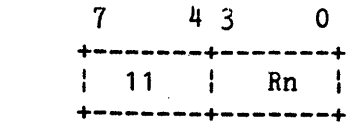
```
OA = Rn + SEXT(displ)          !if byte or word displacement
     or
     Rn + displ                !if longword displacement

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.

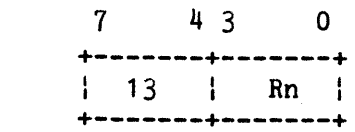
The assembler notation (See Appendix B) for byte, word, and long displacement mode is B^D(Rn), W^D(Rn), and L^D(Rn) respectively where D = displ.

3.4.7 Displacement Deferred Mode

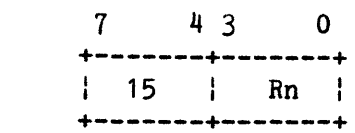
There are 3 operand specifier formats:



The specifier extension is a signed byte displacement. This is termed byte displacement deferred mode.



The specifier extension is a signed word displacement. This is termed word displacement deferred mode.



The specifier extension is a longword displacement. 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:

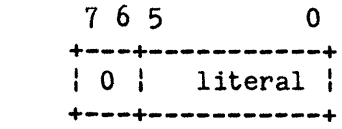
```
OA = (Rn + SEXT(displ))          !if byte or word displacement
     or
     (Rn + displ)                !if longword  displacement

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 (See Appendix B) 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:



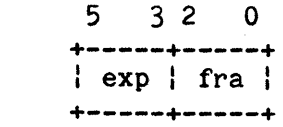
No specifier extension follows.

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

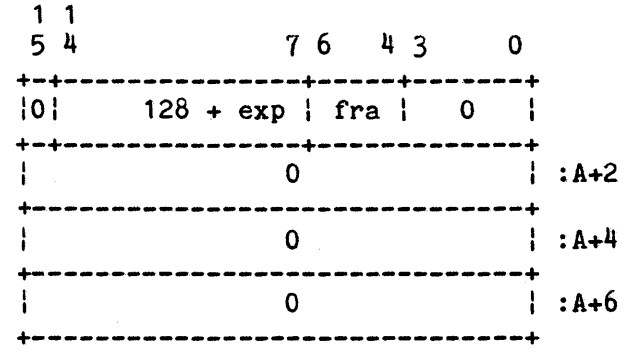
operand = ZEXT(literal)

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, D_floating, G_floating, and H_floating, the 6-bit literal field is composed of 2 3-bit fields:

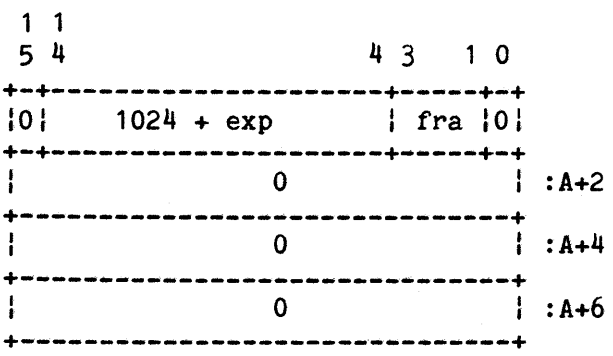


where exp is exponent and fra is fraction. The exp and fra fields are used to form a F_floating or D_floating operand as follows:

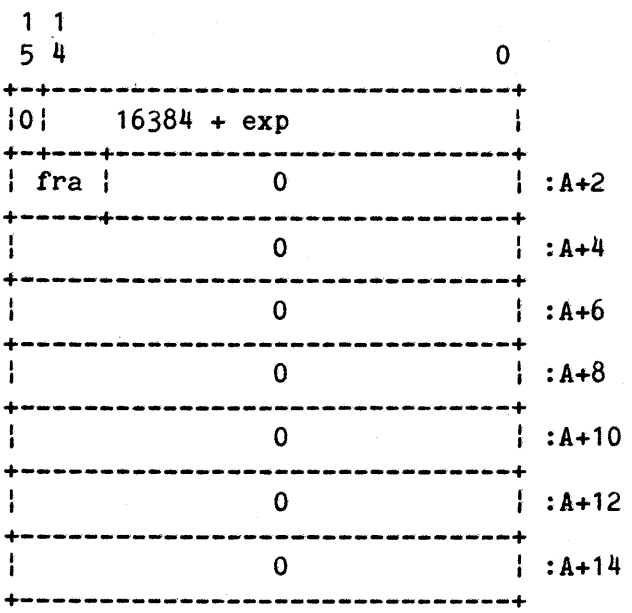


where bits 63:32 are not present in a F_floating operand.

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



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



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

E \ F -->	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

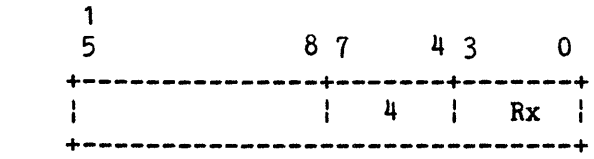
Because there is no operand address, literal mode addressing may not be used for operand specifiers of address access type. Literal mode addressing may also not be used for operand specifiers of write or modify access type. If literal mode is used for operand specifiers of either address, modify, or 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 0 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 (See Appendix B) for literal mode is S^#literal.

3.4.9 Index Mode

The operand specifier format is:



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 (See Appendix B). 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]

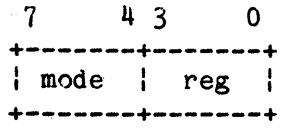
4. autodecrement indexed - -(Rn)[Rx]

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]

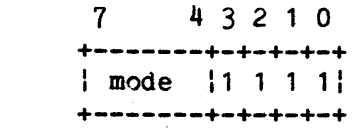
3.5 SUMMARY OF GENERAL MODE ADDRESSING

3.5.1 General Register Addressing



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)



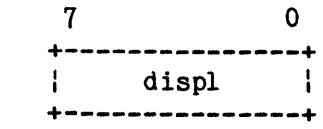
Hex	Dec	Name	Assembler	r	m	w	a	v	PC	SP	Indexable?
8	8	immmediate	I^#constant	y	u	u	y	y	-	-	y
9	9	absolute	@#address	y	y	y	y	y	-	-	y
A	10	byte relative	B^address	y	y	y	y	y	-	-	y
B	11	byte relative deferred	@B^address	y	y	y	y	y	-	-	y
C	12	word relative	W^address	y	y	y	y	y	-	-	y
D	13	word relative deferred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

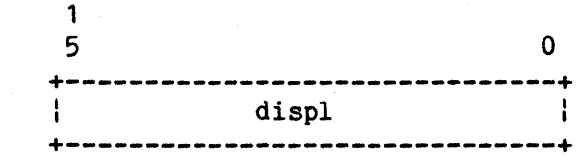
3.6 BRANCH MODE ADDRESSING FORMATS

There are 2 operand specifier formats:



1.

The operand specifier is a signed byte displacement.



2.

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 (See Appendix B) 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 F).

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 4.)

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.

[End of Chapter 3]

Title: VAX-11 Integer and Logical Instructions -- Rev 5

Specification Status: Fully approved

Architectural Status: under ECO control

File: SR4AR5.RNO

PDM #: not used

Date: 27-Oct-78

Superseded Specs: Rev 4

Author: W. Strecker

Typist: B. Call

Reviewer(s): R. Blair, R. Brender, D. Cane, K. Chapman, P. Conklin, D. Cutler, R. Grove, T. Hastings, D. Hustvedt, J. Leonard, P. Lipman, M. Payne, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 4 describes the instructions generally used by all software across all implementations of the VAX-11 architecture. For convenience of review and editting, chapter 4 is separated into a number of specifications. This specification contains the introductory material and the integer arithmetic and logical instructions.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Initial distribution of SRM	Strecker	25-Sep- 75
Rev 2	ECOs 1-11	Strecker	16-Mar-76
Rev 3	ECOs 12-18, April Meeting, and May 25 Meeting	Strecker	10-Jun-76
Rev 4	ECO's	Strecker	12-Mar-77
Rev 5	Octaword ECO	Bhandarkar	27-Oct-78

Rev 4 to Rev 5:

1. Add G_floating and H_floating
2. Add Octaword
3. Add MOVO and CLRO(same as CLRH)

Rev 3 to Rev 4:

1. Typos.
2. Add ADAWI.

Rev 2 to Rev 3:

1. Reserved operand aborts become faults
2. C <- 0 for ASHL, ASHQ
3. Change pointer to longword or address; make it 32 bits
4. Add MINU function in ISP
5. Explicitly give SEXT or ZEXT in all cases needed
6. Add MOVQ, CLRQ
7. Add MOVZBW
8. Change CMP condition codes per ECO 17
9. Change results on overflow in EDIV
10. Specified condition codes on all exceptions
11. Split into separate specifications

Rev 1 to Rev 2:
(* indicated instruction operands or operations changed)

1. Qualified operand names used in instruction description
2. BLISS relational operators used in instruction operation
3. Detailed operation descriptions included on all instructions
4. Condition code settings specified for all instructions
5. PUSH {B,W,F,D} eliminated
6. POP {B,W,L,F,D} eliminated

7. EXCH {B,W,L,F,D} eliminated

8. MOVM {B,W,L,F,D} eliminated

9. MOVN {B,W,L,F,D} changed to MNEG {B,W,L,F,D}

10. MOVC {B,W,L} changed to MCOM {B,W,L}

11. MOVZ {B,W} changed to MOVZ {BW,BL,WL}

12. INS {B,W,L} eliminated

13. ADWC {B,W} eliminated

14. SBWC {B,W} eliminated

15. C <- 0 for MUL {B,W,L}

16. MULX changed to EMUL*

17. DIVX changed to EDIV*

18. MOD {B,W,L} eliminated

19. BFC eliminated

20. AND {B,W,L} eliminated

21. ASH {B,W} eliminated, ASHQ added

22. USH {B,W,L} eliminated

23. ROT {B,W} eliminated

24. ROTC {B,W,L} eliminated

25. SXT {B,W,L} eliminated

26. ADDC {F,D}, SUBC{F,D}, MULC{F,D}, DIVC{F,D} eliminated

27. INC {F,D} eliminated

28. DEC {F,D} eliminated

29. MI {F,D} changed to EMOD {F,D}*

30. CVTR{F,D} L added

31. CHOP{F,D} added

32. POLY{F,D} added

33. MOVA{B,W,L=F,D} changed to MOVP{B,W,L=F,D=Q}

34. PUSHA{B,W,L=F,D} changed to PUSHP{B,W,L=F,D=Q}

35. CMPA{B,W,L=F,D} changed to CMPA*

36. ADTA, SBFA, DIFA added

37. MOVZV changed to EXTZV*

38. EXTV added

39. CMPV changed to CMPZV*

40. CMPV added

41. AVP eliminated

42. FFS/FFC added

43. BPN, BME added

44. BR changed BRB

45. BRW added

46. BSP eliminated

47. BSB changed to BSBB

48. BSBW Added

49. BLS/BLC added

50. CASE{B,W} {B,W,L} eliminated

51. CASE{B,W,L} added*

52. CALL eliminated

53. CALLS/CALLG added

54. RETURN changed to RET*

55. AOB changed to AOBLE

56. AOBLT added

57. SCB eliminated, ACB* redefined

58. SOB changed to SOBGE

59. SOBGT added

60. THRD eliminated

61. JMP/JSB added

62. LSTZ/LSTO eliminated

63. MOVPSW added

64. ASP eliminated

65. HALT added

66. MS{B,W,L,F,D,Q} added

67. MPS{B,W,L=F,D=Q} added

68. ISP added for string instrucions

69. MOVLJS changed to MOVC3/MOVC5

70. MOVRJS eliminated

71. MOVTS changed to MOVTC

72. CMPLRS changed to CMPC3/CMPC5

73. CMPRLS eliminated

74. SCNLRS changed to SCANC

75. SCNRLS eliminated

76. LOCLRS changed to LOCC

77. LOCRLS eliminated

78. EDITS changed to \TBS\

79. MOVTUC proposal added

80. SPANC proposal added

81. MATCHC added

82. MOVN changed to MOVS/MOVU

83. 6 operand ADDN added

84. 6 operand SUBN added

85. 6 operand MULN added

86. 6 operand DIVN Added

87. MODN eliminated

88. CVTLN changed to CVTLS/CVTLU

89. CVTPN changed to CVTPS/CVTPU

90. CVTSN/CVTUN changed to ASHS/ASHU

91. Numeric suffix added to indicate number of operands (e.g. ADDB3)

[End of SR4AR5.RNO]

CHAPTER 4

INSTRUCTIONS

27-Oct-78 -- Rev 5

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 F. The instructions which may be subsetted in certain VAX-11 implementations are listed in Appendix E. Details of the assembler notation are given in Appendix B.

4.1.1 Instruction Descriptions

The instruction set is divided into 12 major sections:

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

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. The notation used to describe operand specifiers is given in Section 4.1.2.

3. The operation of the instruction given in a concise notation. The notation is described briefly in Section 4.1.3.

4. The effect on condition codes. The same notation is used as in the operation description.

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. Additional examples are given in Appendix D.

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.

   w - Operand is written only.

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

   b - byte

   d - double floating or D_floating

   f - floating or 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 Section 3.3. \Ultimately, the notation used here will be described formally or it will be replaced by ISP. In either event a formal description of the notation used to describe instructions will appear in Appendix G.\

+ - 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

MOV Move

Format:

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

Operation:

```
dst <- src;
```

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

The destination operand is replaced by the source operand.

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.

CLR Clear

Format:

opcode dst.wx

Operation:

dst <- 0;

Condition Codes:

N <- 0;
Z <- 1;
V <- 0;
C <- C;

Exceptions:

none

Opcodes:

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

Description:

The destination operand is replaced by 0.

Notes:

CLRx dst is equivalent to MOVx S^#0, dst, but is 1 byte shorter.

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.

MCOM Move Complemented

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

MOVZ Move Zero-Extended

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

CVT Convert

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

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

Description:

The source operand is converted to the data type of the destination operand and the destination operand is replaced by the result. Conversion of a shorter data type to a longer 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.

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.

TST Test

Format:

```
opcode src.rx
```

Operation:

```
src - 0;
```

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

Notes:

TSTx src is equivalent to CMPx src, S^#0, but is 1 byte shorter.

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.

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.

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.

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. A more formal descripion is \TBS\.

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.

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.

SBWC Subtract With Carry

Format:

opcode sub.rl, dif.ml

Operation:

dif <- dif - sub - C;

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

D9	SBWC	Subtract With Carry

Description:

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

Notes:

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

2. The 2 subtractions in the operation are performed simultaneously. A more formal description is \TBS\.

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.

EMUL Extended Multiply

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcodes:

7A	EMUL	Extended Multiply

Description:

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

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 <- {integer overflow} OR {divr EQL 0};
C <- 0;
```

Exceptions:

integer overflow
divide by zero

Opcodes:

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

Description:

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

Notes:

1. Division is performed 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 0.

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 0, 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 0;
Z <- quo EQL 0;
V <- {integer overflow} OR {divr EQL 0};
C <- 0;
```

Exceptions:

```
integer overflow
divide by zero
```

Opcodes:

7B	EDIV	Extended Divide

Description:

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

Notes:

1. The division is performed such that the remainder operand (unless it is 0) 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 0, then the quotient operand is replaced by bits 31:0 of the dividend operand, and the remainder operand is replaced by 0.

ASH Arithmetic Shift

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

78	ASHL	Arithmetic Shift Long
79	ASHQ	Arithmetic Shift Quad

Description:

The source operand is arithmetically shifted by the number of bits specified by the count operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand shifts to the left bringing 0s 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 0 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 0.

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.

BIT Bit Test

Format:

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

Operation:

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

Conditon Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

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 0;
Z <- dst EQL 0;
V <- 0;
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.

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

Exceptions:

none

Opcodes:

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

Description:

In 2 operand format, the 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.

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.

ROTL Rotate Long

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcodes:

9C	ROTL	Rotate Long

Description:

The source operand is rotated logically by the number of bits specified by the count operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand rotates to the left. A negative count operand rotates to the right. A 0 count operand replaces the destination operand with the source operand.

CHAPTER 4

INSTRUCTIONS

31-Jan-79 -- Rev 5

```
************************************************************
*                                                          *
*   THIS SECTION CONTAINS NEW INSTRUCTIONS                 *
*                                                          *
*   FOR EXTENDED RANGE FLOATING POINT DATA TYPES           *
*                                                          *
*      ********** COMPANY CONFIDENTIAL **********         *
*                                                          *
************************************************************
```

Title: VAX-11 Floating Point Instructions -- Rev 5

Specification Status:

Architectural Status: under ECO control

File: SR4BR5.RNO

PDM #: not used

Date: 31-Jan-79

Superseded Specs: Rev 4

Author: W. Strecker

Typist: B. Call

Reviewer(s): R. Blair, R. Brender, D. Cane, K. Chapman, P. Conklin, D. Cutler, R. Grove, T. Hastings, D. Hustvedt, J. Leonard, P. Lipman, M. Payne, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 4 describes the instructions generally used by all software across all implementations of the VAX-11 architecture. For convenience of review and editting, chapter 4 is separated into a number of specifications. This specification contains the floating point instructions.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Initial distribution of SRM	Strecker	25-Sep- 75
Rev 2	ECOs 1-11	Strecker	16-Mar-76
Rev 3	ECOs 12-18, April Meeting, and May 25 Meeting	Strecker	10-Jun-76
Rev 4	ECO to POLY, and ECO to facilitate high speed fl. pt.	Strecker	24-Mar-77
Rev 5	Extended range data types ECO	Bhandarkar	31-Jan-79

Rev 4 to Rev 5

1. Add following instructions:

50FD MOVG Move G_floating

70FD MOVH Move H_floating

7CFD CLRH Clear H_floating

7CFD CLRO Clear Octa

52FD MNEGG Move Negated G_floating

72FD MNEGH Move Negated H_floating

4CFD CVTBG Convert Byte to G_floating

6CFD CVTBH Convert Byte to H_floating

4DFD CVTWG Convert Word to G_floating

6DFD CVTWH Convert Word to H_floating

4EFD CVTLG Convert Long to G_floating

6EFD CVTLH Convert Long to H_floating

48FD CVTGB Convert G_floating to Byte

68FD CVTHB Convert H_floating to Byte

49FD CVTGW Convert G_floating to Word

69FD CVTHW Convert H_floating to Word

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

56FD CVTGH Convert G_floating to H_floating

76FD CVTHG Convert H_floating to G_floating

33FD CVTGF Convert G_floating to Floating

F6FD CVTHF Convert H_floating to Floating

99FD CVTFG Convert Floating to G_floating

98FD CVTFH Convert Floating to H_floating

F7FD CVTHD Convert H_floating to Dfloating

32FD CVTDH Convert Dfloating to H_floating

51FD CMPG Compare G_floating

71FD CMPH Compare H_floating

53FD TSTG Test G_floating

73FD TSTH Test H_floating

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

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

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

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

54FD EMODG Extended Multiply and Integerize G_floating

74FD EMODH Extended Multiply and Integerize H_floating

55FD POLYG Polynomial Evaluation G_floating

75FD POLYH Polynomial Evaluation H_floating

2. Add floating underflow/overflow faults

Rev 3 to Rev 4:

1. Remove destination operand specifier for POLY
2. Reverse the the order of coefficients for POLY
3. POLYF will not set R4 or R5
4. Change the name of second operand of POLY to degree
5. For POLY give reserved operand fault if degree operand > 31.
6. Define certain pathological addressing combinations to give UNPREDICTABLE results to facilitate high speed floating point implementations.
7. Add introduction and rounding/accuracy.
8. POLYx UNPREDICTABLE whether reserved operand if arg = 0.
9. POLYx coefficient reserved can give FPD=0 fault.

Rev 2 to Rev 3:

1. Reserved operand aborts become faults
2. Specify 0 divisor behavior for DIVF, DIVD
3. Change pointer to longword or address; make it 32 bits
4. Add MINU function in ISP
5. Explicitly give SEXT or ZEXT in all cases needed
6. MOVF, MOVD take reserved operand fault
7. Remove round bit
8. Floating overflow, underflow get reserved, 0 respectively
9. Specified condition codes on all exceptions
10. Remove CHOPF, CHOPD
11. Update EMODF, EMODD per ECO 18

12. Change EMODD to produce longword fraction

13. Split into separate specifications

Rev 1 to Rev 2:

See CH4A for changes

[End of SR4BR5.RNO]

4.3 FLOATING POINT INSTRUCTIONS

\The floating point architecture was revised in January 1979 to add two new data types (G_floating and H_floating) and to take faults instead of traps on floating exceptions for the original data types (F_floating and D_floating) as well as the two new data types. Modified versions of the VAX-11/780 with the revised architecture might be built.\

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.

4.3.1 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 0, 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 F 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. Since a reserved operand has a biased exponent of zero, it can be (internally) generated only if overflow or divide by zero occurs on the original VAX-11/780. The action of the VAX-11 in these circumstances is described at the end of the next section.

4.3.2 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. On the original VAX-11/780, floating point instructions trap when floating overflow occurs, and DIV generates a trap if the divisor is 0. For either of these exceptions, a reserved operand is returned as the result, and the N and V condition code bits are set. On all other VAX processors, floating point instructions 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. If the FU bit is set, a trap (on the original VAX-11/780) or fault (on all other VAX processors) occurs on underflow. If underflow occurs, zero is always returned as the result on the original VAX-11/780. On all other VAX processors, zero is returned as the result on underflow only if FU is clear. 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.

4.3.3 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 traps (on the original VAX-11/780) or faults (on all other VAX processors).

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.

It will now be shown that an overflow bit and two guard bits are adequate to guarantee accuracy of rounded ADD, SUB, MUL, or DIV, provided, of course, that the algorithms are properly chosen. Note, first, that ADD or SUB may result in propagation of a carry, and hence the overflow bit is necessary. Second, if in ADD or SUB there is a one bit loss of significance in conjunction with an alignment shift of two or more bits, the first guard bit is needed for the LSB of the normalized result, and the second is then the rounding bit. So the three bits are necessary. A number of constraints must be observed in selection of the algorithms for the basic operations, in order for these three bits to be sufficient to guarantee an error bound of (1/2) LSB:

1. ADD or SUB:

   1. If the alignment shift does not exceed 2 there are no constraints, because no bits can be lost.

   2. If the alignment shift exceeds 2 (or however many guard bits are used, say g GEQ 2), no negations may be made after the alignment shift takes place.

   3. If the above constraint is observed, the error bound for a rounded result is (1/2) LSB. If, however, a negation follows the alignment shift, the error bound will be

      (1/2)*(1 + 2**(-g+2))LSB

      because a "borrow" will be lost on an implicit subtraction, if non-zero bits were lost in the alignment shift. Note that the error bound is 1 LSB if the constraint is ignored and there are only two guard bits (g = 2).

   4. The constraint on no negations after the alignment shift may be replaced by keeping track of non-zero bits lost during the alignment shift, and then negating by one's complement if any "ones" were lost, and by two's complement

if none were lost. If this is done, the error bound will be (1/2) LSB.

2. MUL:

    1. The product of two normalized binary fractions can be as small as 1/4 and must be less than one. The overflow bit is not needed for MUL, but the first guard bit will be necessary for normalization if the product is less than 1/2, and, in this case, the second guard bit is the rounding bit.

    2. The first constraint on MUL is that the product be generated from the least to the most significant bit. Low order bits, in positions to the right of the second guard bit, may be discarded, but ONLY AFTER they have made their contribution to carries which could propagate into the guard bits or beyond.

    3. For the same reasons as for ADD or SUB, if low order bits of the product have been discarded, no negations can be made after generating the product.

3. DIV:

    1. For standard algorithms it is necessary that the remainder be generated exactly at each step; the overflow and two guard bits are adequate for this purpose. The register receiving the quotient must, of course, have a guard bit for the rounding bit, and the quotient must be developed to include the rounding bit.

    2. The Newton-Raphson quadratic convergence algorithms, which might make good use of high speed multiplication logic, require a number of guard bits equal to twice the number of bits desired in the result if the correctness of the rounding bit is to be guaranteed.

The VAX-11 observes all constraints and generates floating point results with an error bound of (1/2) LSB for all floating point instructions except EMOD and POLY. The error analysis of EMOD and POLY is given with their descriptions.

MOV Move

Format:

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

Operation:

```
dst <- src;
```

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

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

Description:

The destination operand is replaced by the source operand.

Notes:

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

CLR Clear

Format:

```
opcode dst.wx
```

Operation:

```
dst <- 0;
```

Condition Codes:

```
N <- 0;
Z <- 1;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

The destination operand is replaced by 0.

Notes:

CLRx dst is equivalent to MOVx #0, dst, but is 5 (F_floating) or 9 (D_floating or G_floating) or 17 (H_floating) bytes shorter.

MNEG Move Negated

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

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

Description:

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

Notes:

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

CVT Convert

Format:

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

Operation:

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

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {src cannot be represented in dst};
C <- 0;
```

Exceptions:

integer overflow
floating overflow
floating underflow
reserved operand

Opcodes:

4C	CVTBF	Convert Byte to F_floating
6C	CVTBD	Convert Byte to D_floating
4CFD	CVTBG	Convert Byte to G_floating
6CFD	CVTBH	Convert Byte to H_floating
4D	CVTWF	Convert Word to F_floating
6D	CVTWD	Convert Word to D_floating
4DFD	CVTWG	Convert Word to G_floating
6DFD	CVTWH	Convert Word to H_floating
4E	CVTLF	Convert Long to F_floating
6E	CVTLD	Convert Long to D_floating
4EFD	CVTLG	Convert Long to G_floating
6EFD	CVTLH	Convert Long to H_floating

48	CVTFB	Convert F_floating to Byte
68	CVTDB	Convert D_floating to Byte
48FD	CVTGB	Convert G_floating to Byte
68FD	CVTHB	Convert H_floating to Byte
49	CVTFW	Convert F_floating to Word
69	CVTDW	Convert D_floating to Word
49FD	CVTGW	Convert G_floating to Word
69FD	CVTHW	Convert H_floating to Word
4A	CVTFL	Convert F_floating to Long
4B	CVTRFL	Convert Rounded F_floating to Long
6A	CVTDL	Convert D_floating to Long
6B	CVTRDL	Convert Rounded D_floating to Long
4AFD	CVTGL	Convert G_floating to Long
4BFD	CVTRGL	Convert Rounded G_floating to Long
6AFD	CVTHL	Convert H_floating to Long
6BFD	CVTRHL	Convert Rounded H_floating to Long
56	CVTFD	Convert F_floating to D_floating
99FD	CVTFG	Convert F_floating to G_floating
98FD	CVTFH	Convert F_floating to H_floating
76	CVTDF	Convert D_floating to F_floating
32FD	CVTDH	Convert D_floating to H_floating
33FD	CVTGF	Convert G_floating to F_floating
56FD	CVTGH	Convert G_floating to H_floating
F6FD	CVTHF	Convert H_floating to F_floating
F7FD	CVTHD	Convert H_floating to D_floating
76FD	CVTHG	Convert H_floating to G_floating

Description:

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

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. On floating overflow, on the original VAX-11/780, the destination operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand), N<-1, Z<-0, V<-1, and C<-0. On all other VAX processors,

floating overflow results in a 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 trap (on the original VAX-11/780) or fault (on all other VAX processors) occurs. On the original VAX-11/780, zero is always stored as the result of floating underflow. On all other VAX processors, 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.

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.

TST Test

Format:

```
opcode src.rx
```

Operation:

```
src - 0;
```

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

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

Description:

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

Notes:

1. TSTx src is equivalent to CMPx src, #0, but is 5 (F_floating) or 9 (D_floating or G_floating) or 17 (H_floating) bytes shorter.

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

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 trap (on the original VAX-11/780) or fault (on all other VAX processors) occurs. On the original VAX-11/780, zero is always stored as the result of floating underflow. On all other VAX processors, 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, on the original VAX-11/780, the sum operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand), N<-1, Z<-0, V<-1, and C<-0. On all other VAX processors, the instruction faults, the sum operand is unaffected, and the condition codes are UNPREDICTABLE.

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 <- {floating overflow};
C <- 0;
```

Exceptions:

floating overflow
floating underflow
reserved operand

Opcodes:

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

Description:

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

Notes:

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

2. On floating underflow, if FU is set a trap (on the original VAX-11/780) or fault (on all other VAX processors) occurs. On the original VAX-11/780, zero is always stored as the result of floating underflow. On all other VAX processors, 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 0 and no exception occurs.

3. On floating overflow, on the original VAX-11/780, the difference operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand), N<-1, Z<-0, V<-1, and C<-0. On all other VAX processors, the instruction faults, the difference 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 0;
Z <- prod EQL 0;
V <- {floating overflow};
C <- 0;
```

Exceptions:

floating overflow
floating underflow
reserved operand

Opcodes:

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

Description:

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

Notes:

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

2. On floating underflow, if FU is set a trap (on the original VAX-11/780) or fault (on all other VAX processors) occurs. On the original VAX-11/780, zero is always stored as the result of floating underflow. On all other VAX processors, 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 0 and no exception occurs.

3. On floating overflow, on the original VAX-11/780, the product operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand), N<-1, Z<-0, V<-1, and C<-0. On all other VAX processors, the instruction faults, the product operand is unaffected, and the condition codes are UNPREDICTABLE.

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 trap (on the original VAX-11/780) or fault (on all other VAX processors) occurs. On the original VAX-11/780, zero is always stored as the result of floating underflow. On all other VAX processors, 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 0 and no exception occurs.

3. On floating overflow, on the original VAX-11/780, the quotient operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand), N<-1, Z<-0, V<-1, and C<-0. On all other VAX processors, 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 0;
Z <- fract EQL 0;
V <- {integer overflow};
C <- 0;
```

Exceptions:

```
integer overflow
floating underflow
reserved operand
```

Opcodes:

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

Description:

The multiplier extension operand is concatenated with the multiplier operand to gain 8 (EMODD and EMODF), 11 (EMODG), or 15 (EMODH) additional low order fraction bits. The low order 5 or 1 bits of the 16-bit multiplier extension operand are ignored by the EMODG and EMODH instructions respectively. The multiplicand operand is multiplied by the extended multiplier operand. The multiplication 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 trap (on the original VAX-11/780) or fault (on all other VAX processors) occurs. On the original VAX-11/780, the integer and fraction parts are replaced by zero on the occurrence of floating underflow. On all other VAX processors, 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 0 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.

POLY Polynomial Evaluation

Format:

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

Operation:

```
tmp1 <- degree;
if tmp1 GTRU 31 then RESERVED OPERAND EXCEPTION;
tmp2 <- tbladdr;
tmp3 <- {(tmp2)+};        !tmp3 accumulates the partial result
                          !tmp3 is of type x
if POLYH then -(SP) <- arg;
tmp4 <- 0;        !underflow flag
while tmp1 GTRU 0 do
        begin             !computation loop
        tmp3 <- {arg * tmp3};
                !Perform multiply, and retain a 31 (POLYF),
                !63 (POLYD, POLYG), or 127 (POLYH) bit fraction.
                !(The fraction is truncated before normalization.)
                !Use the result in the following add operation.
        tmp3 <- tmp3 + (tmp2);
                !normalize,round, and check for over/underflow
                !only after the combined multiply/add sequence
        if OVERFLOW then FLOATING OVERFLOW EXCEPTION
        if UNDERFLOW then
                begin
                if FU EQL 1 then FLOATING UNDERFLOW FAULT;
                                !except for original VAX-11/780
                tmp3 <- 0;      !force result to 0;
                tmp4 <- 1;      !set underflow flag
                end;
        tmp1 <- tmp1 - 1;
        tmp2 <- tmp2 + {size of data type};
        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 <- 0;
        end;
if POLYH then
        begin
```

```
                SP <- SP + 16;
                R3'R2'R1'R0 <- tmp3;
                R4 <- 0;
                R5 <- tmp2;
                end;
        if tmp4 EQL 1 then FLOATING UNDERFLOW TRAP
```

! only on the original VAX-11/780

AND FU EQL 1

Condition Codes:

```
        N <- R0 LSS 0;
        Z <- R0 EQL 0;
        V <- {floating overflow};
        C <- 0;
```

Exceptions:

floating overflow
floating underflow
reserved operand

Opcodes:

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

Description:

The table address operand points to a table of polynomial coefficients. The coefficient of the highest order term of the polynomial is pointed to by the table address operand. The table is specified with lower order coefficients stored at increasing addresses. The data type of the coefficients is the same as the data type of the argument operand.

The evaluation is carried out by Horner's method and the contents of R0 (R1'R0 for POLYD and POLYG, R3'R2'R1'R0 for POLYH)) are replaced by the result. The result computed is:

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

The unsigned word degree operand specifies the highest numbered coefficient to participate in the evaluation.

POLYH requires four longwords on the stack to store arg in case the instruction is interrupted.

Notes:

1. After execution:

   POLYF

   R0 = result
   R1 = 0
   R2 = 0
   R3 = table address + degree*4 + 4

   POLYD and POLYG

   R0 = high order part of result
   R1 = low order part of result
   R2 = 0
   R3 = table address + degree*8 + 8
   R4 = 0
   R5 = 0

   POLYH

   R0 = highest order part of result
   R1 = second highest order part of result
   R2 = second lowest order part of result
   R3 = lowest order part of result
   R4 = 0
   R5 = table address + degree*16 + 16

2. On a floating fault:

   1. PSL<FPD> = 0 and the instruction faults, or

   2. PSL<FPD> = 1 and the instruction is suspended. State is saved in the general registers as follows:

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

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

      POLYH
      R3'R2'R1'R0 = partial result after iteration prior to the
                    one causing the overflow/underflow
      R4<7:0> = tmp1 !number of iterations remaining

R4<31:8> = implementation specific
R5 = tmp2 !points to table entry causing exception
arg is on previous mode stack.

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. The multiplication is performed such that ~~the precision of the product is equivalent to a floating point datum having~~ a 31 bit (POLYF), 63 bit (POLYD and POLYG), or 127 bit (POLYH) fraction is retained.

4. If the unsigned word degree operand is 0, the result is C[0].

5. If the unsigned word degree operand is greater than 31, a reserved operand exception occurs.

6. On a reserved operand exception:

   1. if PSL<FPD> = 0, 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.

7. On floating underflow after the rounding operation, the temporary result (tmp3) is replaced by zero (always on the original VAX-11/780; and only if FU is clear on all other VAX processors), and the operation continues. On the original VAX-11/780, if FU is set a floating underflow trap occurs at the end of the instruction if underflow occurred during any iteration of the computation loop. Note that the final result may be non zero if underflow occurred before the last iteration. On all other VAX processors, if FU is set a floating underflow fault occurs immediately.

8. On floating overflow after the rounding operation at any iteration of the computation loop, the instruction terminates and causes a trap (on the original VAX-11/780) or fault (on all other VAX processors). On overflow traps the contents of R2 and R3 are UNPREDICTABLE for POLYF, ~~POLYD, and POLYG~~, and zero for POLYH; the contents of ~~R4~~ and ~~R5~~ are UNPREDICTABLE for POLYD, ~~POLYG, and POLYH;~~ R0 contains the reserved operand (minus 0) and R1 = 0.

9. POLY can have both over- and underflow in the same instruction. If both occur, the overflow exception is taken; underflow is lost.

10. 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.

11. For POLYH, some implementations may not push 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.

Example:

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

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

If any coef, other than c[0] ≠ 0, then...

degree x, c[1], others

CHAPTER 4

INSTRUCTIONS

30-Nov-78 -- Rev 5

```
************************************************************
*                                                          *
*   THROW THIS PAGE AWAY.                                  *
*                                                          *
*   This is the third part of Chapter 4.                   *
*                                                          *
*          ***** COMPANY CONFIDENTIAL ****                 *
*                                                          *
************************************************************
```

Title: VAX-11 Miscellaneous and Control Instructions -- Rev 5

Specification Status: Fully Approved

Architectural Status: under ECO control

File: SR4CR5.RNO

PDM #: not used

Date: 30-Nov-78

Superseded Specs: Rev 4

Author: W. Strecker

Typist: B. Call

Reviewer(s): R. Blair, R. Brender, D. Cane, K. Chapman, P. Conklin, D. Cutler, R. Grove, T. Hastings, D. Hustvedt, J. Leonard, P. Lipman, M. Payne, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 4 describes the instructions generally used by all software across all implementations of the VAX-11 architecture. For convenience of review and editting, chapter 4 is separated into a number of specifications. This specification contains the address, variable length bit field, control, miscellaneous, and queue instructions.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Initial distribution of SRM	Strecker	25-Sep- 75
Rev 2	ECOs 1-11	Strecker	16-Mar-76
Rev 3	ECOs 12-18, April Meeting, and May 25 Meeting	Strecker	10-Jun-76
Rev 4	ECO's	Strecker	31-Mar-77
Rev 5	ECO's	Bhandarkar	30-Nov-78

Rev 4 to Rev 5:

1. Add following instructions:

   7EFD MOVAH Move Address H_floating

   7EFD MOVAO Move Address Octa

   7FFD PUSHAH Push Address H_floating

   7FFD PUSHAO Push Address Octa

   4FFD ACBG Add Compare and Branch G_floating

   6FFD ACBH Add Compare and Branch H_floating

2. Fix variable bit field description

3. Add interlocked self-relative queue instructions

4. Change INSV condition codes

   1. INSQHI

   2. INSQTI

   3. REMQHI

   4. REMQTI

Rev 3 to Rev 4:

1. Correct Opcode Assignments.

2. CF to FP (FP ECO).

3. CALL Standard ECO.

4. Add INDEX instruction (INDEX ECO).

5. Expand descriptions of XFC, HALT, and BPT.

6. Motivate call frame, etc.

7. Clear T on CALL (T bit ECO).

8. Reverse incorrect opcode assignments of FFC, FFS.

9. Note that FFC/FFS conditon codes had been changed between rev 2 and rev 3.

Rev 2 to Rev 3:

1. Reserved operand aborts become faults
2. Change BLS and BLC to BLBS and BLBC
3. Change number of bytes referenced on access to zero length field from 1 to zero
4. Change HALT in non-kernel mode to privileged instruction fault
5. Change BPT to Breakpoint fault
6. Change MOVPSW TO MOVPSL
7. Add Queue instructions
8. Change pointer to longword or address; make it 32 bits
9. Add MINU function in ISP
10. Explicitly give SEXT or ZEXT in all cases needed
11. Specified condition codes on all exceptions
12. Remove CMPA, DIFA, ADTA, SBFA
13. Include pos<31> in Field
14. Correct operand type typos in Field
15. Change conditional branches per ECO 17
16. Add BBSSI, BBCCI
17. In ACBx, SOBxx, AOBxx set N, Z, V from the add
18. Change names to AOBLEQ, AOBLSS, SOBGEQ, SOBGTR
19. Specify branch behavior on overflow ACBx, AOBxxx, SOBxxx
20. In CALLS, CALLG remove DZ, FV; set DV, IV from entry mask; clear FU; remove ring crossing
21. BISPSW, BICPSW takes reserved operand fault
22. Remove MSx, MSPx
23. Split into separate specifications

Rev 1 to Rev 2:

See CH4A for changes

[End of SR4CR5.RNO]

4.4 ADDRESS INSTRUCTIONS

MOVA Move Address

Format:

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

Operation:

```
dst <- src;
```

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

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 <- (SP) LSS 0;
Z <- (SP) EQL 0;
V <- 0;
C <- C;

Exceptions:

none

Opcodes:

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

Description:

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

Notes:

1. 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.

4.5 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.

EXT Extract Field

Format:

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

Operation:

```
dst <- if size NEQU 0 then SEXT(FIELD(pos, size, base))
                      else 0;          !EXTV

dst <- if size NEQU 0 then ZEXT(FIELD(pos, size, base))
                      else 0;          !EXTZV
```

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

EE	EXTV	Extract Field
EF	EXTZV	Extract Zero-Extended Field

Description:

For EXTV, the destination operand is replaced by the sign extended field specified by the position, size, and base operands. For EXTZV, the destination operand is replaced by the zero extended field specified by the position, size and base operands. If the size operand is 0, the only action is to replace the destination operand with 0 and affect the condition codes.

Notes:

1. A reserved operand fault occurs if:

    1. size GTRU 32.

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

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

INSV Insert Field

Format:

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

Operation:

```
if size NEQU 0 then FIELD(pos, size, base) <-
        src<{size - 1}:0>;
```

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

F0	INSV	Insert Field

Description:

The field specified by the position, size, and base operands is replaced by bits size-1:0 of the source operand. If the size operand is 0, 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 0, and the field is contained in the registers.

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

CMP Compare Field

Format:

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

Operation:

```
tmp <- if size NEQU 0 then SEXT(FIELD (pos,
       size, base)) else 0;    !CMPV
tmp - src;

tmp <- if size NEQU 0 then ZEXT(FIELD (pos,
       size, base)) else 0;    !CMPZV
tmp - src;
```

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

EC	CMPV	Compare Field
ED	CMPZV	Compare Zero-Extended Field

Description:

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

Notes:

1. A reserved operand fault occurs if:

    1. size GTRU 32.

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

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

FF Find First

Format:

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

Operation:

```
state = if {FFS} then 1 else 0;
if size NEQU 0 then
        begin
        tmp1 <- FIELD(startpos, size, base);
        tmp2 <- 0;
        while {tmp1<tmp2> NEQ state} AND
                {tmp2 LEQU {size - 1}} do
                tmp2 <- tmp2 + 1;
        findpos <- startpos + tmp2;
        end
else
        findpos <- startpos;
```

Condition Codes:

```
N <- 0;
Z <- {bit not found};
V <- 0;
C <- 0;
```

Exceptions:

reserved operand

Opcodes:

EB	FFC	Find First Clear
EA	FFS	Find First Set

Description:

A field specified by the start position, size, and base operands is extracted. The field is tested for a bit in the state indicated by the instruction starting at bit 0 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 0, 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, 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.

4.6 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.

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

   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 0
BGEQU   C EQL 0
BGTRU   {C OR Z} EQL 0
```

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

3. Signed Group

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

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

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.

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.

BB Branch on Bit

Format:

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

Operation:

```
teststate = if {BBSS or BBSC} then 1 else 0;
newstate = if {BBSS or BBCS} then 1 else 0;
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.ab, 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 0;
if src<0> 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 0) 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.

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, the index operand is replaced by 0 (always on the original VAX-11/780 and only if FU is clear on all other VAX processors); and comparison and branch determination proceed normally. On the original VAX-11/780, a trap occurs at the end of the instruction if FU is set. On all other VAX processors, a fault occurs if FU is set and the index operand is unaffected.

4. On floating overflow, on the original VAX-11/780, the index operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand); N<-1, Z<-0, V<-1; and the branch is not taken. On all other VAX processors, 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 0;
Z <- index EQL 0;
V <- {integer overflow};
C <- C;
```

Exceptions:

integer overflow

Opcodes:

F3	AOBLEQ	Add One and Branch Less Than or Equal

Description:

One is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If 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.

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 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:

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 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.

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.

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.

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:

05	RSB	Return From Subroutine

Description:

PC is replaced by a longword popped from the stack.

Notes:

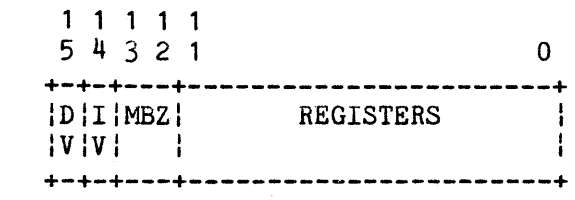
1. RSB is used to return from subroutines called by the BSBB, BSBW and JSB instructions.

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

4.7 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 Appendix C 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.

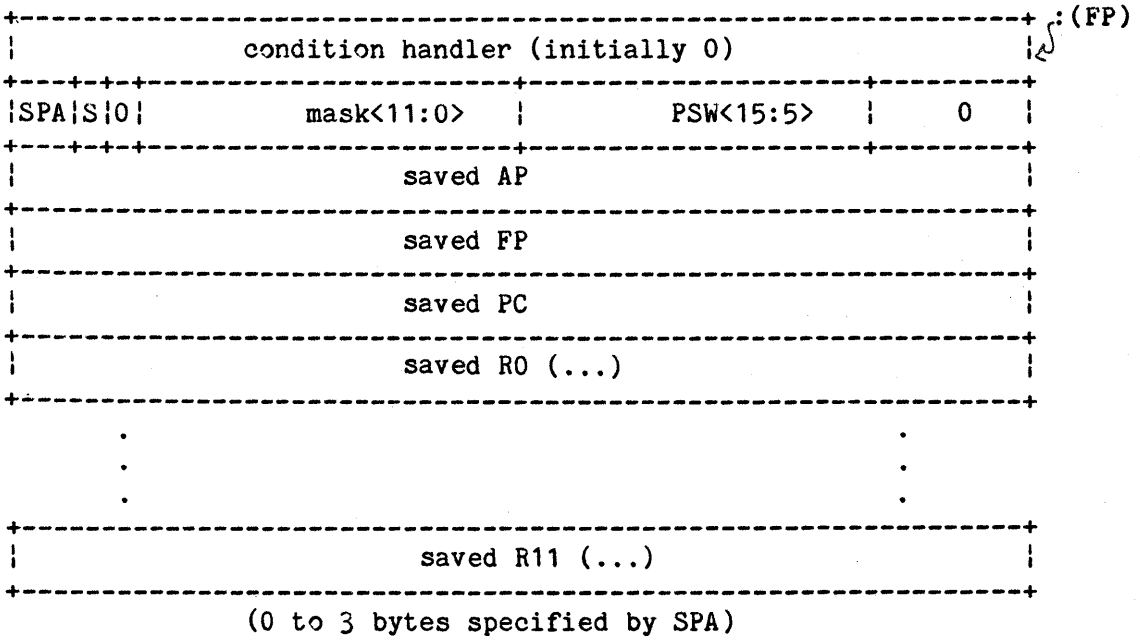
The entry mask specifies the subprocedure's register use and overflow enables:



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 R0 specified by bits 11 through 0 respectively are saved on the stack and are restored by the RET instruction. The procedure calling standard requires that all registers in the range R2 through R11 used in the procedure must appear in the mask. In addition, the CALL instructions always preserve PC, SP, FP, and AP. Thus, a procedure can be considered as equivalent to a complex instruction which stores a value into R0 and R1, affects memory, and clears the condition codes. If the procedure has no function value, the contents of R0 and R1 can be considered as UNPREDICTABLE.

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:



(0 to 3 bytes specified by SPA)

S = set if CALLS; clear if CALLG.

Note that the saved condition codes are cleared. The contents of the frame PSW<3:0> at the time RET is executed will become the condition codes resulting from the execution of the procedure. Similarly, the saved trace enable (PSW<T>) is cleared.

CALLG Call Procedure With General Argument List

Format:

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

Operation:

```
tmp2 <- (dst);
tmp1 <- SP;
SP<1:0> <- 0;
for tmp3 <- 11 step -1 until 0 do
        if tmp2<tmp3> EQL 1 then
        -(SP) <- R[tmp3];
-(SP) <- PC;
-(SP) <- FP;
-(SP) <- AP;
PSW<N,Z,V,C> <- 0;
-(SP) <- tmp1<1:0>'0'0'tmp2<11:0>'PSW<15:5>'0<4:0>;
-(SP) <- 0;
FP <- SP;
AP <- arglist;
PSW<DV> <- tmp2<15>;
PSW<IV> <- tmp2<14>;
PSW<FU> <- 0;
PC <- dst + 2;
```

Condition Codes:

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

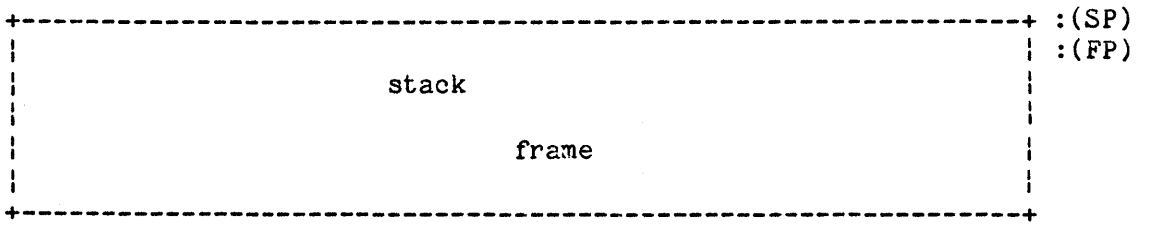
reserved operand

Opcodes:

FA	CALLG	Call Procedure with General Argument List

Description:

SP is saved in a temporary and then bits 1:0 are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to 0 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 31:30, a 0 in bit 29 and bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0 with T cleared is pushed on the stack. A longword 0 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.



(0 to 3 bytes specified by SPA)

Notes:

1. If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs.

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

3. The procedure calling standard and the condition handling facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers R2 through R11 which are modified in the called procedure must be preserved in the mask. Refer to Appendix C.

CALLS Call Procedure with Stack Argument List

Format:

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

Operation:

```
tmp2 <- (dst);
-(SP) <- numarg;
tmp1 <- SP;
SP<1:0> <- 0;
for tmp3 <- 11 step -1 until 0 do
        if tmp2<tmp3> EQL 1 then
        -(SP) <- R[tmp3];
-(SP) <- PC;
-(SP) <- FP;
-(SP) <- AP;
PSW<N,Z,V,C> <- 0;
-(SP) <- tmp1<1:0>'1'0'tmp2<11:0>'PSW<15:5>'0<4:0>;
-(SP) <- 0;
FP <- SP;
AP <- tmp1;
PSW<DV> <- tmp2<15>;
PSW<IV> <- tmp2<14>;
PSW<FU> <- 0;
PC <- dst + 2;
```

Condition Codes:

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

reserved operand

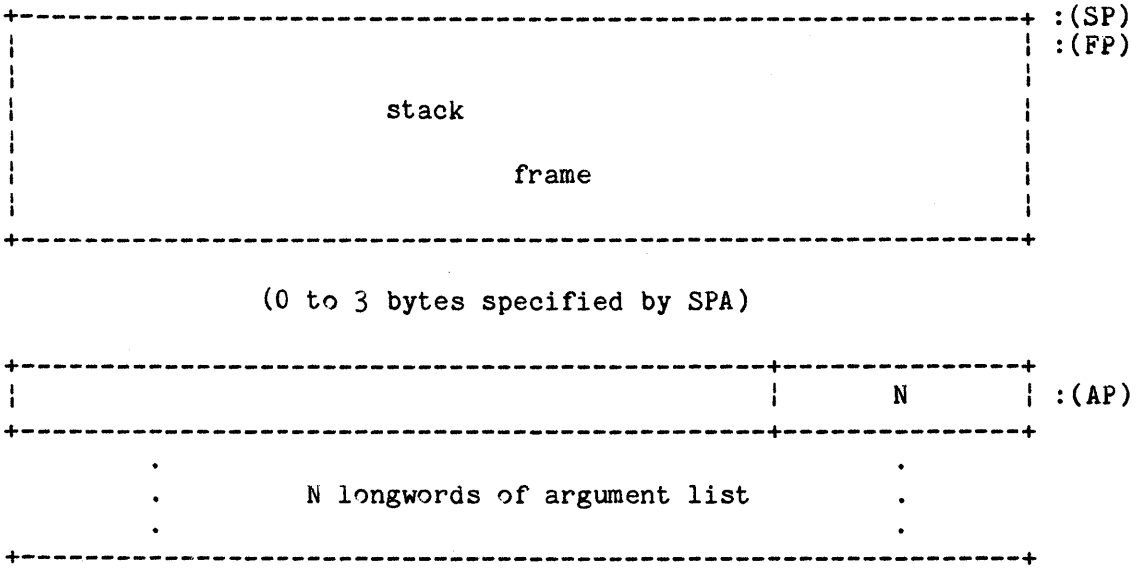
Opcodes:

FB	CALLS	Call Procedure With Stack Argument List

Description:

The numarg operand is pushed on the stack as a longword (byte 0 contains the number of arguments, high order 24 bits are used by DIGITAL software). SP is saved in a temporary and then bits 1:0 of SP are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit 0 and the contents of registers whose 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 31:30, a 1 in bit 29, a 0 in bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0 with T cleared is pushed on the stack. A longword 0 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:



Notes:

1. If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs.

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

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

4. The procedure calling standard and the condition handling facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers R2 through R11 which are modified in the called procedure must be preserved in the entry mask. Refer to Appendix C.

RET Return from Procedure

Format:

```
opcode
```

Operation:

```
SP <- FP + 4;
tmp1 <- (SP)+;
AP <- (SP)+;
FP <- (SP)+;
PC <- (SP)+;
tmp2 <- tmp1<27:16>;
for tmp3 <- 0 step 1 until 11 do
        if tmp2<tmp3> EQL 1 then
        R[tmp3] <- (SP)+;
SP <- SP + ZEXT{tmp1<31:30>}
PSW <- tmp1<15:0>;
if tmp1<29> EQL 1 then
        begin
        tmp4 <- 4 * ZEXT({(SP)+}<7:0>);
        SP <- SP + tmp4;
        end;
```

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

04	RET	Return from Procedure

Description:

SP is replaced by FP plus 4. A longword containing stack alignment bits in bits 31:30, a CALLS/CALLG flag in bit 29, the low 12 bits of the procedure entry mask in bits 27:16, and a saved PSW in bits 15:0 is popped from the stack and saved in a temporary. PC, CF, 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 0 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 31:30 of the temporary. PSW is replaced by bits 15:0 of the temporary. If bit 29 in the temporary is 1 (indicating that

the procedure was called by CALLS), a longword containing the number of arguments is popped from the stack. Four times the unsigned value of the low byte of this longword is added to SP and SP is replaced by the result.

Notes:

1. A reserved operand fault occurs if tmp1<15:8> NEQ 0.

2. On a reserved operand fault, the condition codes are UNPREDICTABLE. The value of tmp1<28> is ignored.

3. The procedure calling standard and condition handling facility assume that procedures which return a function value or a status code do so in R0 or R0 and R1. See Appendix C.

4.8 MISCELLANEOUS INSTRUCTIONS

BPT Breakpoint Fault

Format:

```
opcode
```

Operation:

```
PSL<TP> <- 0;
{breakpoint fault};
```

Condition Codes:

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes:

03	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 degugging 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;
V <- 0;
C <- 0;

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.

XFC Extended Function Call

Format:

```
opcode
```

Operation:

```
{XFC fault};
```

Condition Codes:

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

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.

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

Descriiption:

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
```

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.

POPR Pop Registers

Format:

```
opcode mask.rw
```

Operation:

```
for tmp <- 0 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 0 to bit 14. Bit 15 is ignored.

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).

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.

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.

NOP No Operation

Format:

opcode

Operation:

none

Condition Codes:

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

Exceptions:

none

Opcodes:

01	NOP	No Operation

Description:

No operation is performed.

4.9 QUEUE INSTRUCTIONS

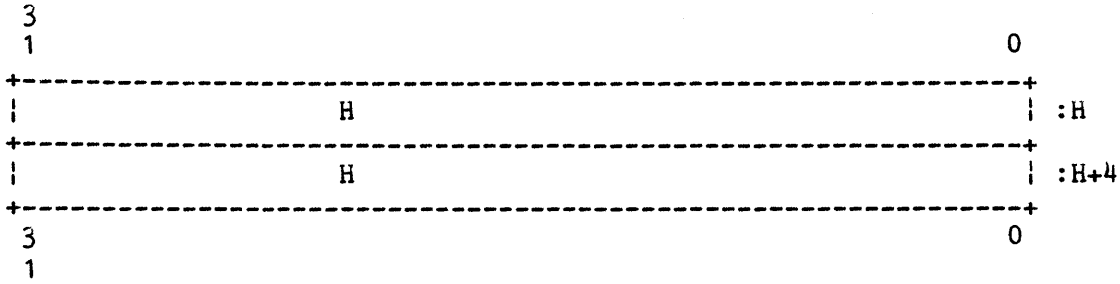
A queue is a circular, doubly linked list. A queue entry is specified by its address. Each queue entry is linked to the next via a pair of longwords. The first longword is the forward link : it specifies the location of the succeeding entry. The second longword is the backward link : it specifies the 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.

4.9.1 Absolute Queues

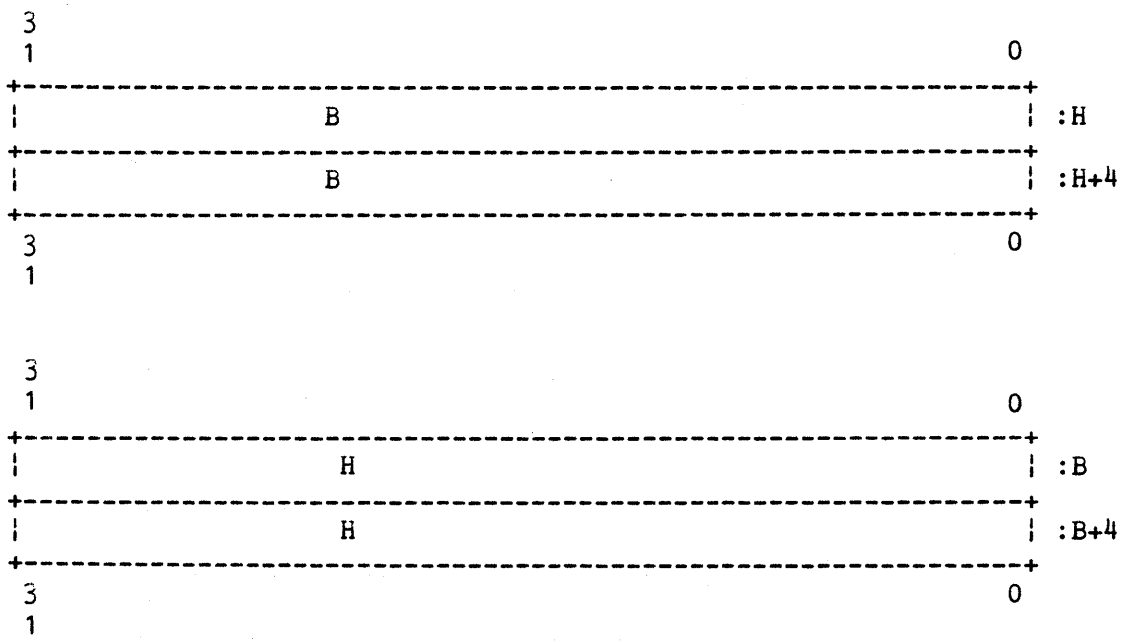
Absolute queues use absolute addresses as links. Queue entries are linked by a pair of longwords. The first (lowest addressed) longword is the forward link: the address of the succeeding queue entry. The second (highest addressed) longword is the backward link: the address of the preceding queue entry.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

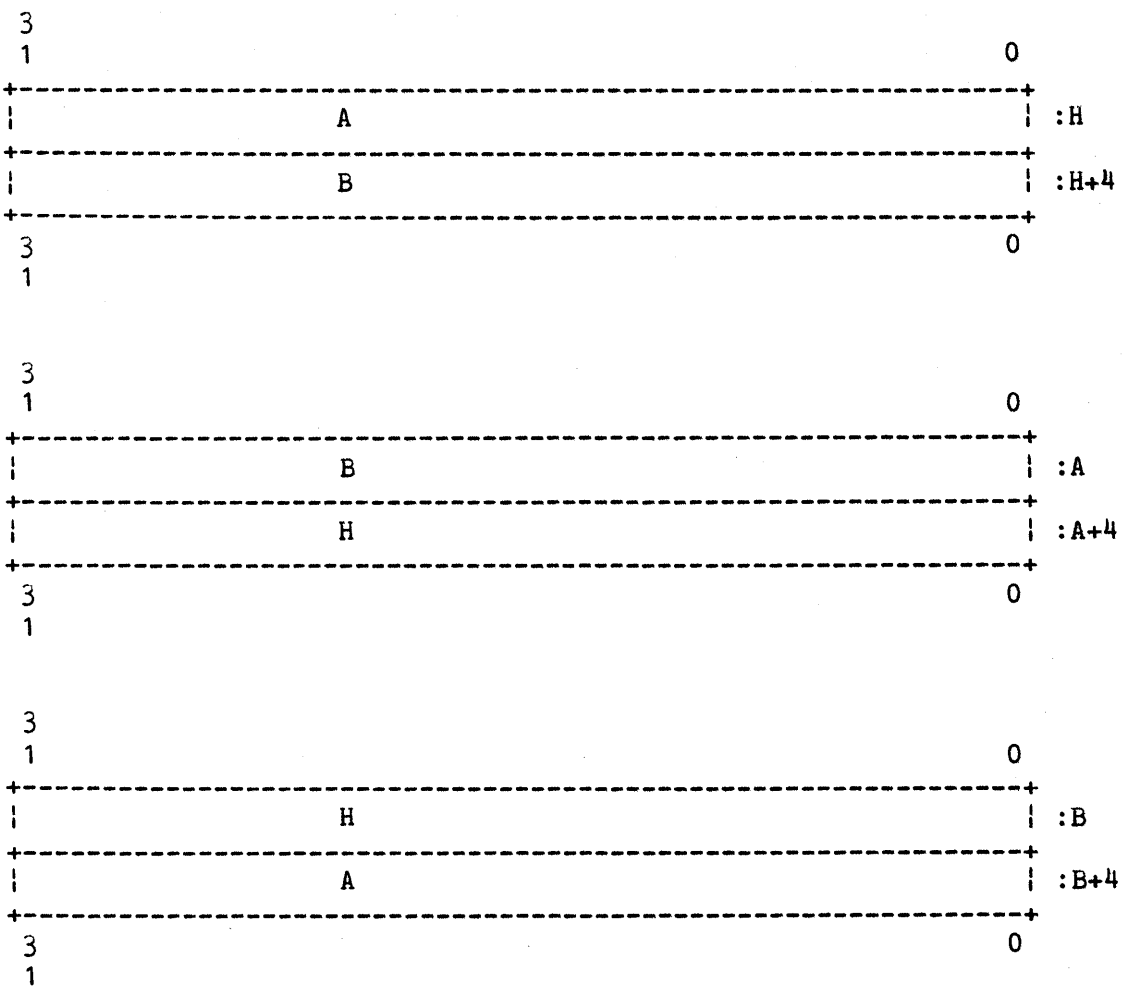
The following contains examples of queue operations. An empty queue is specified by its header at address H:



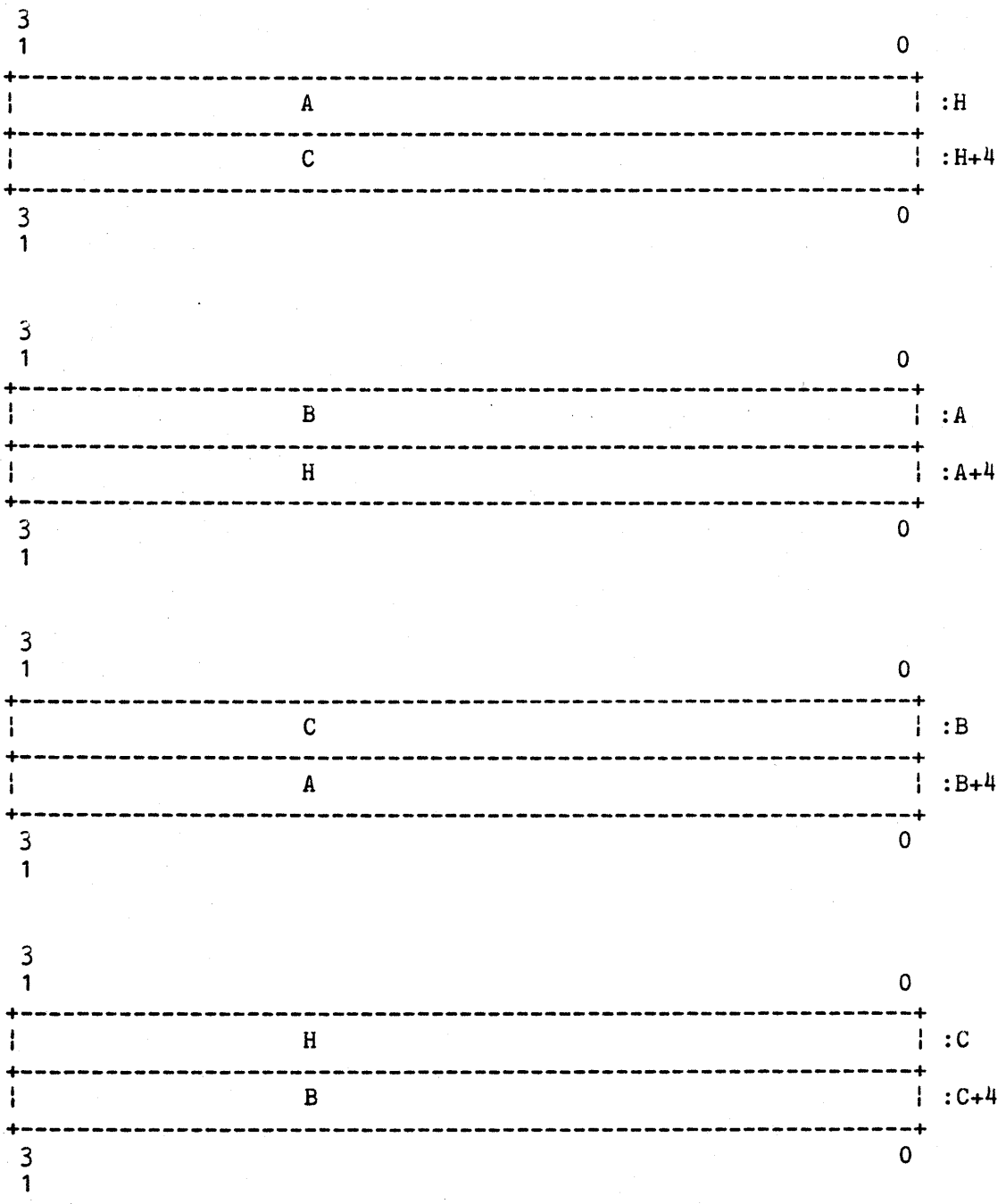
If an entry at address B is inserted into an empty queue (at either the head or tail), the queue is as shown below:



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

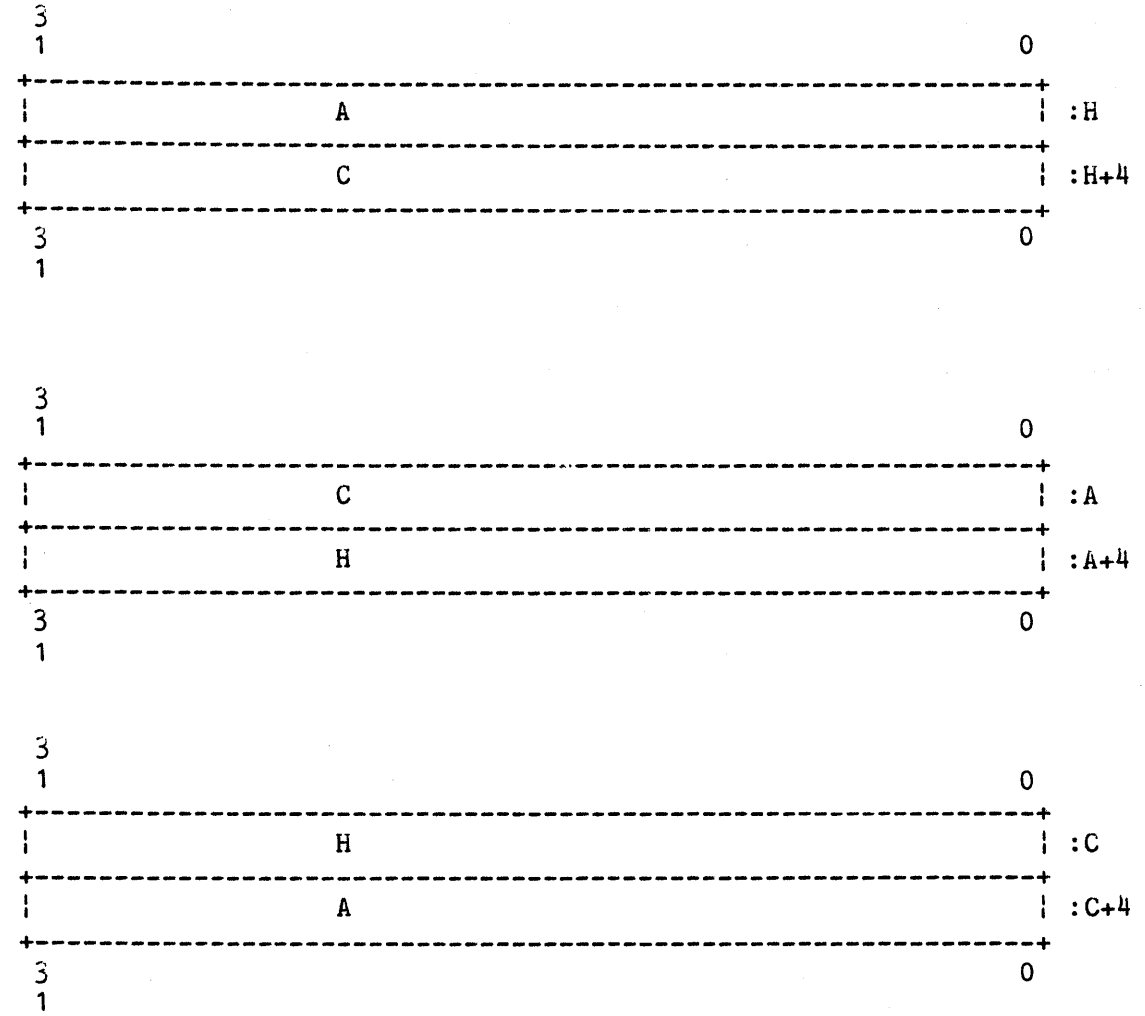


Finally, if an entry at address C is inserted at the tail, the queue appears as follows:



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:



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.

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 <- 0;
C <- (entry) LSSU (entry+4);
```

Exceptions:

none

Opcodes:

OE 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.

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:

OF	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:

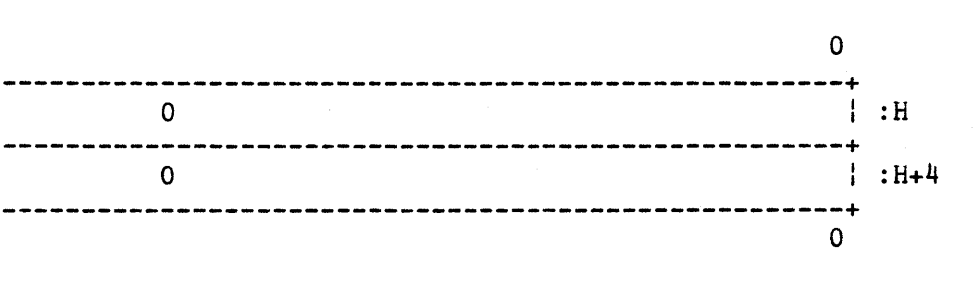
   ```
   1$:  REMQUE  ...             ;anything removed?
        BVS     EMPTY           ;no
          .
          .
          .
        BR      1$              ;
   ```

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

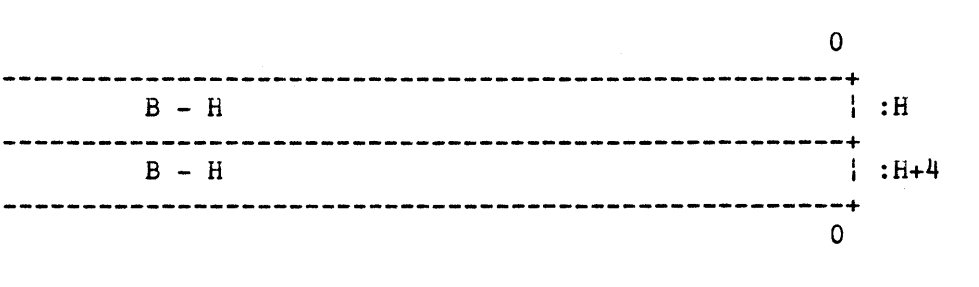
4.9.2 Self-relative Queues

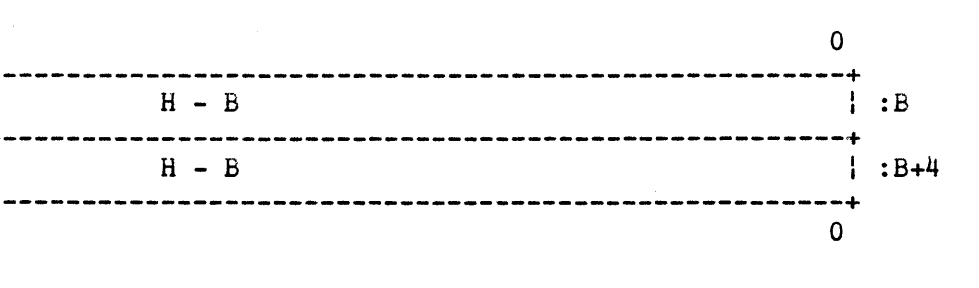
Self-relative queues use displacements from queue entries as links. Queue entries are linked by a pair of longwords. The first longword (lowest addressed) is the forward link : displacement of the succeeding queue entry from the present entry. The second longword (highest addressed) is the backward link: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

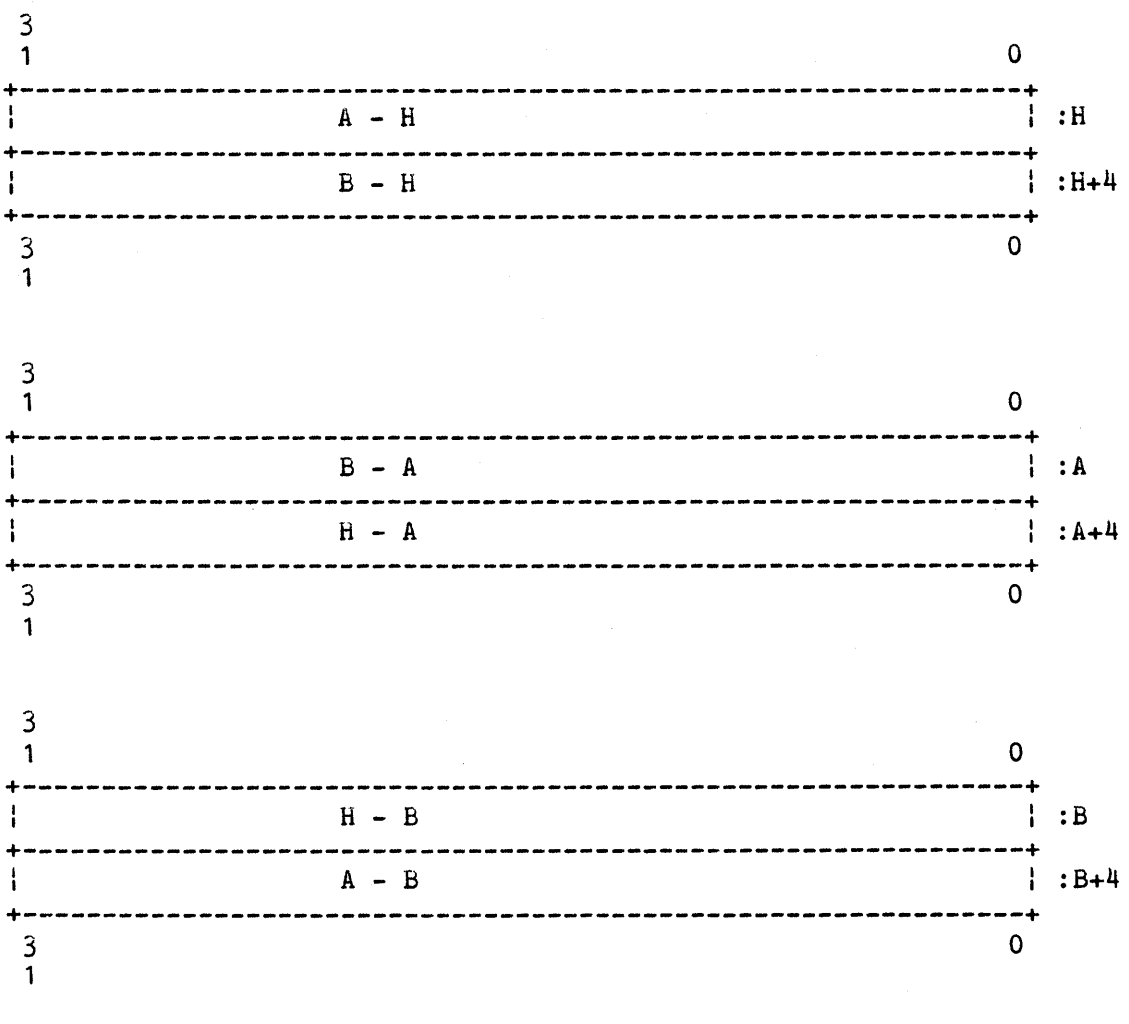


If an entry at address B is inserted into an empty queue (at either the head or tail), the queue is as shown below:

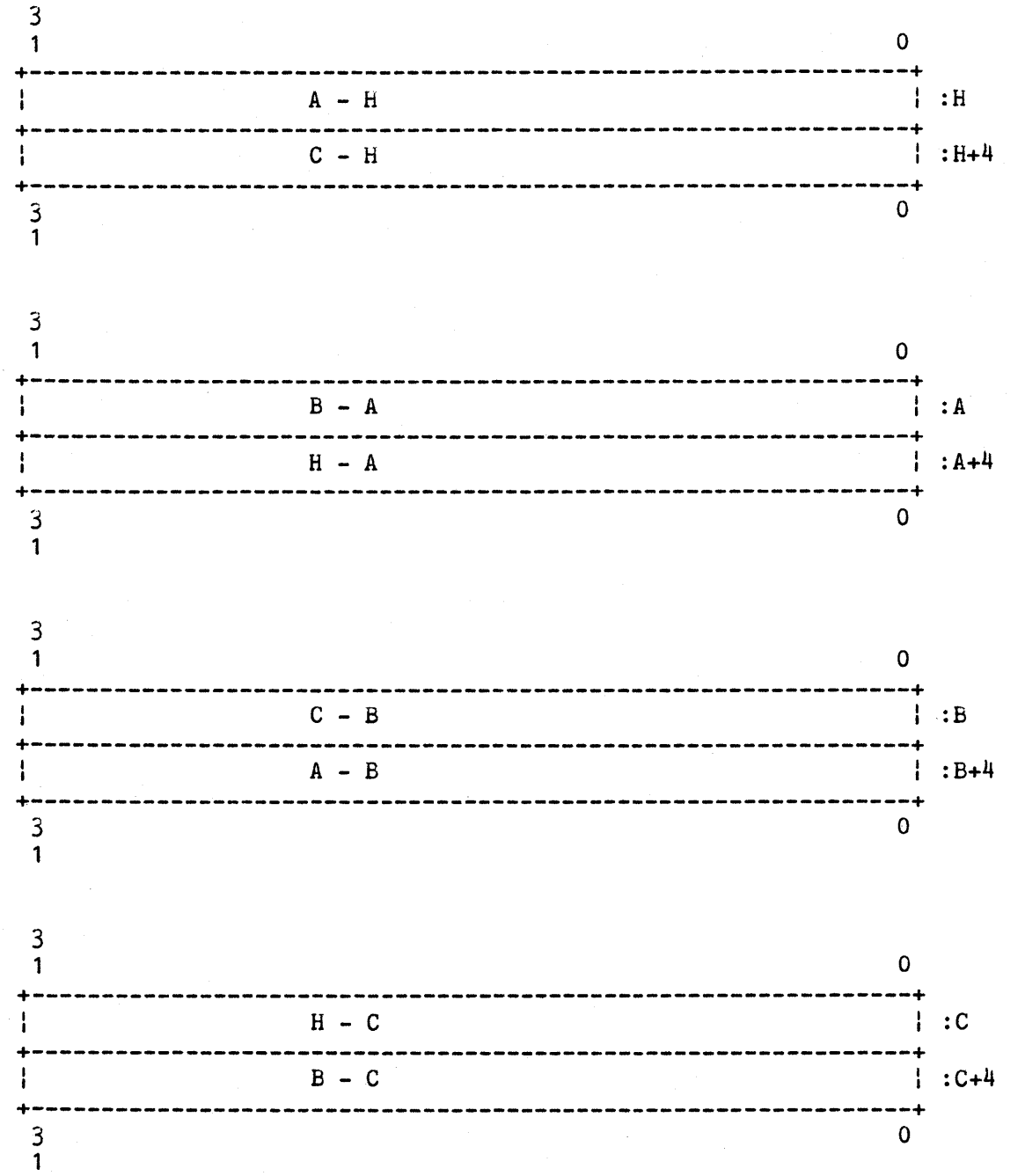




If an entry at address A is inserted at the head of the queue, the queue is as shown below:



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



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 0 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.

INSQHI Insert Entry into Queue at Head, Interlocked

Format:

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

Operation:

```
tmp1 <- (header){interlocked};  !acquire hardware interlock
                                !must have write access to header
                                !header must be quadword aligned
                                !header cannot be equal to entry
                                !tmp1<2:1> must be zero

if tmp1<0> EQLU 1 then
        begin
        (header){interlocked} <- tmp1; !release hardware interlock
        {set condition codes and terminate instruction};
        end;
else
        begin
        (header){interlocked} <- tmp1 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 + tmp1
                !Also, check for quadword alignment
                begin
                tmp2 <- header - entry;
                (entry) <- tmp1 + tmp2;     !forward link of entry
                (entry + 4) <- tmp2;        !backward link of entry
                (header + tmp1 + 4) <- -tmp1 - tmp2;
                                           !backward link of successor
                {read (header){interlocked}};
                                           !acquire hardware interlock
                (header){interlocked} <- -tmp2;
                        !forward link of header, release interlocks
                end;
        else
                begin
                {read (header){interlocked}};
                                !acquire hardware interlock
                (header){interlocked} <- tmp1;
                                           !release all interlocks
                {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. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals entry. In this case the queue is not altered.

INSQTI Insert Entry into Queue at Tail, Interlocked

Format:

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

Operation:

```
tmp1 <- (header){interlocked};  !acquire hardware interlock
                                !must have write access to header
                                !header must be quadword aligned
                                !header cannot be equal to entry
                                !tmp1<2:1> must be zero

if tmp1<0> EQLU 1 then
        begin
        (header){interlocked} <- tmp1;  !release hardware interlock
        {set condition codes and terminate instruction};
        end;
else
        begin
        (header){interlocked} <- tmp1 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
                tmp2 <- (header + 4);
                tmp3 <- header - entry;
                (entry) <- tmp3;        !forward link of entry
                (entry + 4) <- tmp2 + tmp3; !backward link of entry
                if {tmp2 NEQU 0} then (header+tmp2) <- -tmp3 - tmp2
                        else tmp1 <- -tmp3 - tmp2;
                                !forward link of predecessor
                (header+4) <- -tmp3; !backward link of header
                {read (header){interlocked}};
                                !acquire hardware interlock
                (header){interlocked} <- tmp1;  !release interlocks
                end;
        else
                begin
                {read (header){interlocked}};
                                !acquire hardware interlock
                (header){interlocked} <- tmp1;  !release interlocks
                {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. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals entry. In this case the queue is not altered.

REMQHI Remove Entry from Queue at Head, Interlocked

Format:

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

Operation:

```
tmp1 <- (header){interlocked};  !acquire hardware interlock
                                !must have write access to header
                                !header must be quadword aligned
                                !header cannot be equal to addr
                                !tmp1<2:1> must be zero

if tmp1<0> EQLU 1 then
        begin
        (header){interlocked} <- tmp1;  !release hardware interlock
        {set condition codes and terminate instruction};
        end;
else
        begin
        (header){interlocked} <- tmp1 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 + tmp1 {if tmp1 NEQU 0}
                !write into header + tmp1 + (header + tmp1) {if
                !                                  tmp1 NEQU 0}
                !Also, check for quadword alignment
                begin
                addr <- header + tmp1;
                if {tmp1 EQL 0} then tmp2 <- header
                        else tmp2 <- addr + (addr);
                (tmp2 + 4) <- header - tmp2;
                                !backward link of successor
                {read (header){interlocked}};
                                !acquire hardware interlock
                (header){interlocked} <- tmp2 - header;
                !forward link of header, release all interlocks
                end;
        else
                begin
                {read (header){interlocked}};
                                !acquire hardware interlock
                (header){interlocked} <- tmp1;  !release interlocks
                {backup instruction};
                {initiate fault};
                end;
        end;
```

Condition Codes:

```
        if {removal succeeded} then
                begin
                N <- 0;
                Z <- (header) EQL 0;     !queue empty
                V <- tmp1 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 Tail, 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. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals addr. In this case the queue is not altered.

REMQTI Remove Entry from Queue at Tail, Interlocked

Format:

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

Operation:

```
tmp1 <- (header){interlocked};  !acquire hardware interlock
                                !must have write access to header
                                !header must be quadword aligned
                                !header cannot be equal to addr
                                !tmp1<2:1> must be zero

if tmp1<0> EQLU 1 then
        begin
        (header){interlocked} <- tmp1;  !release hardware interlock
        {set condition codes and terminate instruction};
        end;
else
        begin
        (header){interlocked} <- tmp1 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 tmp1
                !                                        NEQU 0}
                !write into header + (header + 4)
                !    + (header + 4 + (header + 4)) {if tmp1 NEQU 0}
                !Also, check for quadword alignment
                begin
                addr <- header + (header + 4);
                tmp2 <- addr + (addr + 4);
                (header + 4) <- tmp2 - header;
                                !backward link of header
                tmp3 <- tmp1;   !save tmp1 to set Z correctly
                if {tmp2 EQL header} then tmp1 <- 0
                        else(tmp2) <- header - tmp2;
                                !forward link of predecessor
                {read (header){interlocked}};
                (header){interlocked} <- tmp1;  !release interlocks
                end;
        else
                begin
                {read (header){interlocked}};
                                !acquire hardware interlock
                (header){interlocked} <- tmp1;  !release interlocks
                {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. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals addr. In this case the queue is not altered.

CHAPTER 4

INSTRUCTIONS

26-Oct-78 -- Rev 5

```
**********************************************************
*                                                        *
*   THROW THIS PAGE AWAY.                                *
*                                                        *
*   This is the fourth part of Chapter 4.                *
*                                                        *
**********************************************************
```

Title: VAX-11 Character String Instructions -- Rev 5

Specification Status: Fully Approved

Architectural Status: under ECO control

File: SR4DR5.RNO

PDM #: not used

Date: 26-Oct-78

Superseded Specs: Rev 4

Author: W. Strecker

Typist: B. Call

Reviewer(s): R. Blair, R. Brender, D. Cane, K. Chapman, P. Conklin, D. Cutler, R. Grove, T. Hastings, D. Hustvedt, J. Leonard, P. Lipman, M. Payne, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 4 describes the instructions generally used by all software across all implementations of the VAX-11 architecture. For convenience of review and editting, chapter 4 is separated into a number of specifications. This specification contains the character string and cyclic redundancy check instructions.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Initial distribution of SRM	Strecker	25-Sep- 75
Rev 2	ECOs 1-11	Strecker	16-Mar-76
Rev 3	ECOs 12-18, April Meeting, and May 25 Meeting	Strecker	10-Jun-76
Rev 4	ECO's	Strecker	31-Mar-77
Rev 5	Editorial, MATCHC ECO	Bhandarkar	26-Oct-78

Rev 4 to Rev 5:

1. Typos
2. MATCHC ECO

Rev 3 to Rev 4:

1. Typos
2. ECO to complete documentation of CRC Instruction.
3. Clarify null strings.
4. Add AUTODIN-II CRC.

Rev 2 to Rev 3:

1. Reserved operand aborts become faults
2. Add SKPC
3. Change pointer to longword or address; make it 32 bits
4. Add MINU function in ISP
5. Explicitly give SEXT or ZEXT in all cases needed
6. Specified condition codes on all exceptions
7. MOVTC does not translate fill
8. Increase registers used by MOVC3 and MOVC5 to 6
9. Change condition codes setting MOVC, MOVTC, MOVTUC
10. Add MOVTUC, MATCHC
11. Add CRC per ECO 12
12. Change CRC table operand from .al to .ab
13. Split into separate specifications

Rev 1 to Rev 2:

See CH4A for changes

[End of SR4DR5.RNO]

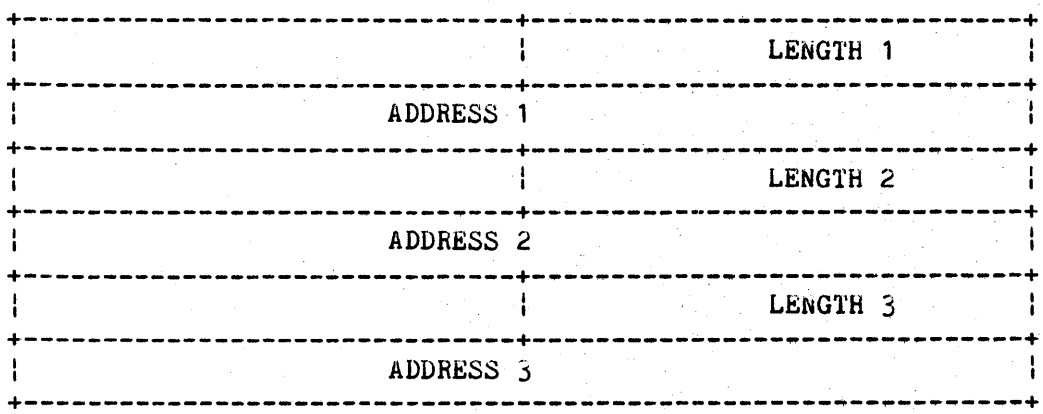
4.10 CHARACTER STRING INSTRUCTIONS

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 R0 through R1, R0 through R3, or R0 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 interrupt, the instruction resumes transparently. The format of the control block is:



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.

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:

```
tmp1 <- len;                                   !3 operand
tmp2 <- srcaddr;
tmp3 <- dstaddr;
if tmp2 GTRU tmp3 then
        begin
        while tmp1 NEQU 0 do
                begin
                (tmp3) <- (tmp2);
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 + 1;
                tmp3 <- tmp3 + 1;
                end;
        R1 <- tmp2;
        R3 <- tmp3;
        end
else
        begin
        tmp4 <- tmp1;
        tmp2 <- tmp2 + ZEXT(tmp1);
        tmp3 <- tmp3 + ZEXT(tmp1);
        while tmp1 NEQU 0 do
                begin
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 - 1;
                tmp3 <- tmp3 - 1;
                (tmp3) <- (tmp2);
                end;
        R1 <- tmp2 + ZEXT(tmp4);
        R3 <- tmp3 + ZEXT(tmp4);
        end;
R0 <- 0;
R2 <- 0;
R4 <- 0;
R5 <- 0;
```

```
        tmp1 <- srclen;                                 !5 operand
        tmp2 <- srcaddr;
        tmp3 <- dstlen;
        tmp4 <- dstaddr;
        if tmp2 GTRU tmp4 then
                begin
                while (tmp1 NEQU 0) AND (tmp3 NEQU 0) do
                        begin
                        (tmp4) <- (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;
                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  0 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;
        R0 <- tmp1;
        R2 <- 0;
        R4 <- 0;
        R5 <- 0;
```

Condition Codes:

```
N <- 0;                     !MOVC3
Z <- 1;
V <- 0;
C <- 0;

N <- srclen LSS dstlen; !MOVC5
Z <- srclen EQL dstlen;
V <- 0;
C <- srclen LSSU dstlen;
```

Exceptions:

none

Opcodes:

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

Description:

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

Notes:

1. After execution of MOVC3:

   R0 = 0

   R1 = address of one byte beyond the source string

   R2 = 0

   R3 = address of one byte beyond the destination string.

   R4 = 0

   R5 = 0

2. After execution of MOVC5:

   R0 = number of unmoved bytes remaining in source string.
   R0 is 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 = 0

   R3 = address of one byte beyond the destination string .

   R4 = 0

   R5 = 0

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

4. MOVC5 with a 0 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;
        R0 <- tmp1;
        R2 <- 0;
        R3 <- tbladdr;
        R4 <- 0;
```

Condition Codes:

```
        N <- srclen LSS dstlen;
        Z <- srclen EQL dstlen;
        V <- 0;
        C <- srclen LSSU dstlen;
```

Exceptions:

none

Opcodes:

2E	MOVTC	Move Translated Characters

Description:

The source string specified by the source length and source address operands is translated 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:

R0 = number of untranslated bytes remaining in source string;
R0 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 = 0

R3 = address of the translation table.

R4 = 0

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 0 and tmp3 GTRU 0 then
        begin

while (tmp1 NEQU 0) AND (tmp3 NEQU 0) 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;

R0 <- tmp1;
R1 <- tmp2;
R2 <- 0;
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 source string or the table, the destination string and registers R0 through R5 are UNPREDICTABLE.

Notes:

After execution:

R0 = number of bytes remaining in source string (including the byte which caused the escape). R0 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 = 0

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.

CMPC Compare Characters

Format:

```
opcode len.rw, src1addr.ab, src2addr.ab  3 operand

opcode src1len.rw, src1addr.ab, fill.rb,
       src2len.rw, src2addr.ab            5 operand
```

Operation:

```
tmp1 <- len;                                  !3 operand
tmp2 <- src1addr;

tmp3 <- src2addr;
if tmp1 EQL 0 then; !Condition Codes affected on tmp1 EQL 0
if tmp1 GTRU 0 then
        begin
while (tmp1 NEQU 0) do
if (tmp2) EQL (tmp3) then
                !Condition Codes affected on ((tmp2) EQL (tmp3))
        begin
        tmp1 <- tmp1 - 1;
        tmp2 <- tmp2 + 1;
        tmp3 <- tmp3 + 1;
        end;
        else exit while loop;
        end;
R0 <- tmp1;
R1 <- tmp2;
R2 <- R0;
R3 <- tmp3;

tmp1 <- src1len;                              !5 operand
tmp2 <- src1addr;
tmp3 <- src2len;
tmp4 <- src2addr;

if (tmp1 EQL 0) AND (tmp3 EQL 0) then;
        !Condition codes affected on (tmp1 EQL 0) AND (tmp3 EQL 0)
while (tmp1 NEQU 0) AND (tmp3 NEQU 0) do
if (tmp2) EQL (tmp4) then
                !Condition Codes affected on ((tmp2) EQL (tmp4))
        begin
        tmp1 <- tmp1 - 1;
        tmp2 <- tmp2 + 1;
        tmp3 <- tmp3 - 1;
        tmp4 <- tmp4 + 1;
        end;
else exit while loop;
if NOT(tmp1 NEQU 0) AND (tmp3 NEQU 0) then
        begin
while (tmp1 NEQU 0) AND ((tmp2) EQL fill) do
```

```
                        !Condition Codes affected on ((tmp2) EQL fill)
                begin
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 + 1;
                end;
        while (tmp3 NEQU 0) AND (fill EQL (tmp4)) do
                        !Condition Codes affected on (fill EQL (tmp4))
                begin
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 + 1;
                end;
                end;
        R0 <- 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 <- 0;
        C <- (first byte) LSSU (second byte);
```

Exceptions:

none

Opcodes:

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

Description:

In 3 operand format, the bytes of 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:

   R0 = number of bytes remaining in string 1 (including byte which terminated comparison); R0 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 = R0

   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:

   R0 = number of bytes remaining in string 1 (including byte which terminated comparison); R0 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.

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:

   R0 = number of bytes remaining in the string (including the byte which produced the non-zero AND result)
   R0 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 = 0

   R3 = address of the table

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

SPANC Span 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) NEQ 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:

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:

   R0 = number of bytes remaining in the string (including the byte which produced the zero AND result) R0 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 = 0

   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.

LOCC Locate Character

Format:

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

Operation:

```
tmp1 <- len;
tmp2 <- addr;
if tmp1 GTRU 0 then
        begin
while (tmp1 NEQ 0) AND ((tmp2) NEQ char)  do
        begin
        tmp1 <- tmp1 - 1;
        tmp2 <- tmp2 + 1;
        end;
        end;
R0 <- tmp1;
R1 <- tmp2;
```

Condition Codes:

```
N <- 0;
Z <- R0 EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes:

3A	LOCC	Locate Character

Description:

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

Notes:

1. After execution:

   R0 = number of bytes remaining in the string (including located one) if byte located; otherwise 0

   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.

SKPC Skip Character

Format:

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

Operation:

```
tmp1 <- len;
tmp2 <- addr;
if tmp1 GTRU 0 then
        begin
while {tmp1 NEQ 0} AND {(tmp2) EQL char} do
        begin
        tmp1 <- tmp1 - 1;
        tmp2 <- tmp2 + 1;
        end;
        end;
R0 <- tmp1;
R1 <- tmp2;
```

Condition Codes:

```
N <- 0;
Z <- R0 EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes:

3B	SKPC	Skip Character

Description:

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

Notes:

1. After execution:

   R0 = number of bytes remaining in the string (including the unequa one) if unequal byte located; otherwise 0

   R1 = address of the byte located if byte located; otherwise addre

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.

MATCHC Match Characters

Format:

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

Operation:

```
tmp1 <- objlen;
tmp2 <- objaddr;
tmp3 <- srclen;
tmp4 <- srcaddr;
tmp5 <- tmp1;
while (tmp1 NEQU 0) AND (tmp3 GEQU tmp1) do
        begin
        if (tmp2) EQL (tmp4) then
                begin
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 + 1;
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 + 1;
                end
        else
                begin
                tmp2 <- tmp2 - ZEXT (tmp5-tmp1);
                tmp3 <- (tmp3 - 1) + (tmp5-tmp1);
                tmp4 <- (tmp4 + 1) - ZEXT (tmp5-tmp1);
                tmp1 <- tmp5;
                end;
        end;
R0 <- tmp1;
R1 <- tmp2;
R2 <- tmp3;
R3 <- tmp4;
```

Condition Codes:

```
N <- 0;
Z <- R0 EQL 0;  !match found
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes:

39 MATCHC Match Characters

Description:

The source string specified by the source length and source address operands is searched for a substring 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:

   R0 = if a match occurred 0; otherwise the number of bytes in the object string.

   R1 = if a match occurred, the address of one byte beyond the object string; otherwise the address of the object string.

   R2 = if a match occurred, the number of bytes remaining in the source string; otherwise 0.

   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.

2. If both strings have zero length or if the object string has zero length, condition code Z is set 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 just as though the substring were not found.

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 0 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 0 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 0 do
                begin
                tmp3<7:0><- tmp3<7:0> 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-1:0>*i)};
                tmp1 <- tmp1 -1;
                end;
        R0 <- tmp3;
        R1 <- 0;
        R2 <- 0;
        R3 <- tmp2;     !address of end of string + 1
```

Condition Codes:

```
        N <- R0 LSS 0;
        Z <- R0 EQL 0;
        V <- 0;
        C <- 0;
```

Exceptions:

none

Opcodes:

0B	CRC	Calculate Cyclic Redundancy Check

Description:

The CRC of the data stream described by the string descriptor is calculated. The initial CRC is given by inicrc and is normally 0 or -1 unless the CRC is calculated in several steps. The result is left in R0. If the polynomial is less than order-32, the result must be extracted from the 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 R0.

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.r1, 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(0:15), TMP, X

      DO 190 INDEX = 0, 15

      TMP = INDEX
      DO 150 I = 1, 4
      X = TMP .AND. 1
      TMP = ISHFT(TMP,-1)     !logical shift right one bit
      IF (X .EQ. 1) TMP =  TMP .XOR. POLY
150   CONTINUE
      TABLE(INDEX) = TMP

190   CONTINUE
      RETURN
      END
```

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:           120001 (octal)
      initialize:     0
      result:         R0<15:0>
```

   CCITT (used in ADCCP, HDLC, SDLC)

```
      polynomial:     x^16 + x^12 + x^5 + 1
      poly:           102010 (octal)
      initialize:     -1<15:0>
```

result: one's complement of R0<15:0>

AUTODIN-II

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

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 [0] is always 0 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<0>	8	[0],[8]
2	4	tmp3<1:0>	4	[0],[4],[8],[12]
4	2	tmp3<3:0>	1	all

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

CHAPTER 4

INSTRUCTIONS

2-Nov-78 -- Rev 5

****
*
* THROW THIS PAGE AWAY.
*
* This is the fifth part of Chapter 4.
*
****

Title: VAX-11 Decimal String Instructions -- Rev 5

Specification Status: Fully Approved

Architectural Status: under ECO control

File: SR4ER5.RNO

PDM #: not used

Date: 2-Nov-78

Superseded Specs: Rev 4

Author: W. Strecker / T. Rarich

Typist: B. Call

Reviewer(s): R. Blair, R. Brender, D. Cane, K. Chapman, P. Conklin, D. Cutler, R. Grove, T. Hastings, D. Hustvedt, J. Leonard, P. Lipman, M. Payne, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 4 describes the instructions generally used by all software across all implementations of the VAX-11 architecture. For convenience of review and editting, chapter 4 is separated into a number of specifications. This specification contains the decimal string instructions.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Initial distribution of SRM	Strecker	25-Sep-75
Rev 2	ECOs 1-11	Strecker	16-Mar-76
Rev 3	ECOs 12-18, April Meeting, and May 25 Meeting	Strecker	10-Jun-76
Rev 4	Packed Decimal Standard Additions to CIS Instructions	Rarich	15-Feb-77
Rev 5	Typos	Bhandarkar	2-Nov-78

Rev 4 to Rev 5:

1. Typos

Rev 3 to Rev 4

1. Change Numeric string to Trailing Numeric
2. Change CVTNP & CVTPN To CVTTP & CVTPT.
3. Add Leading Separate String converts CVTSP,CVTPS
4. ASHP now treats -0 overflow like all other instructions.
5. MULP & DIVP now produce +0 unless decimal overflow occurs.
6. All operations now produce +0 unless decimal overflow occurs.
7. Define reserved operand and overflow handling once at beginning of section.
8. Add note on convert to/from longword to discuss use of registers and overlap.
9. Add clarifying notes at the beginning of the section on zero length strings etc.
10. Add note on overlap with CMPP instruction.
11. Change zoned to packed operations (decimal data ECO).
12. Clarify -0 in all results and sources (decimal data ECO attachment).
13. Clarify overflow as non-zero digits, not significant digits.
14. Leave CVTLP address in R3 not R1.
15. ASHP changed to include rounding operand.

Rev 2 to Rev 3:

1. Reserved operand aborts become faults
2. Add empty section for EDITN
3. Change pointer to longword or address; make it 32 bits
4. Add MINU function in ISP
5. Explicitly give SEXT or ZEXT in all cases needed

6. Specified condition codes on all exceptions

7. Lower limit on decimal string length is 0

8. Digits not checked by numeric instructions

9. Remove MULN4, DIVN4; change names to MULN, DIVN

10. Clobber R2, R3 in CVTLN, CVTNL

11. CVTNL returns correctly signed low order bits of result on overflow

12. Remove CVTLU, CVTPU, ASHU; change names of CVTLS, CVTPS, ASHS to CVTLN, CVTPN, ASHN

13. Split into separate specifications

Rev 1 to Rev 2:

See CH4A for changes

[End of SR4ER5.RNO]

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.

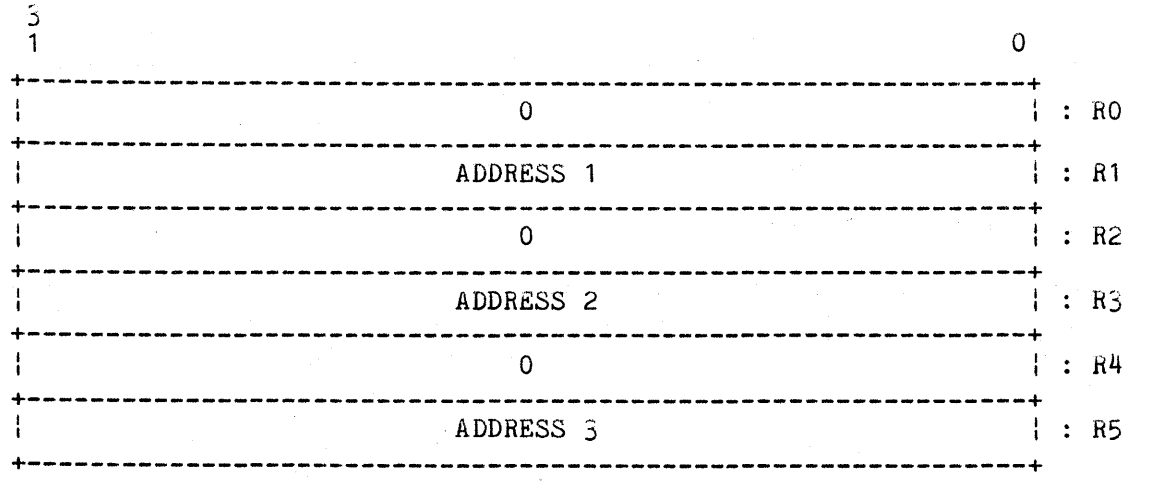
A decimal string is specified by 2 operands:

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 R0 through R3 or R0 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:



The fields ADDRESS 1, ADDRESS 2 and ADDRESS 3 (if required) contain the address of the byte containing the lowest addressed byte. in 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 non-zero digits of the result. On overflow, the destination string is replaced by the correctly signed least significant digits of the result (even if the result is -0). Note that neither the high nibble of an even length packed decimal string, nor the sign byte of a Leading Separate Numeric string is used to store result digits.

4.12.2 Zero Numbers

A zero result has a positive sign for all operations which complete without decimal overflow. However, when digits are lost because of overflow, a zero result receives the sign (positive or negative) of the correct result.

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

4.12.3 Reserved Operand Exception

A reserved operand fault occurs if the length of a decimal string operand is outside the range 0 through 31, or if an invalid sign or digit is encountered in CVTSP, and CVTTP.

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 fault 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 "0" 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 0. In this case, the value is zero (plus or minus) and one byte of storage is occupied. This byte must contain a "0" digit in the high nibble and the sign in the low nibble.

The length of a trailing numeric string can be 0. 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 length of a Leading Separate Numeric string can be 0. 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 fault will occur if an invalid sign is detected. The value of a zero length decimal string is identically 0.

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 0;
Z <- {dst string} EQL 0;
V <- 0;
C <- C;
```

Exceptions:

reserved operand

Opcodes:

34 MOVP Move Packed

Description:

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

Notes:

1. After execution:

   R0 = 0

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

   R2 = 0

   R3 = address of the byte containing the most significant digit of the destination string.

2. The destination string, R0 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 fault occurs.

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

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 = 0

R3 = address of the byte containing the most significant digit of string 2.

2. R0 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 fault occurs.

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:

   R0 = 0

   R1 = address of the byte containing the most significant digit of the addend string

   R2 = 0

   R3 = address of the byte containing the most significant digit of the sum string

2. After execution of ADDP6:

   R0 = 0

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

   R2 = 0

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

   R4 = 0

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

3. The sum string, R0 through R3 (or R0 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 fault occurs.

4. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.

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:

   R0 = 0

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

   R2 = 0

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

2. After execution of SUBP6:

   R0 = 0

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

   R2 = 0

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

   R4 = 0

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

3. The difference string, R0 through R3 (R0 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 fault occurs.

4. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.

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 0;
Z <- {prod string} EQL 0;
V <- {decimal overflow};
C <- 0;
```

Exceptions:

```
reserved operand
decimal overflow
```

Opcodes:

```
  25    MULP    Multiply Packed
```

Description:

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

Notes:

1. After execution:

   R0 = 0

   R1 = address of the byte containing the most significant digit of the multiplier string

   R2 = 0

   R3 = address of the byte containing the most significant digit of the multiplicand string

   R4 = 0

R5 = address of the byte containing the most significant digit of the product string

2. The product string, R0 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 fault occurs.

3. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.

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 ~~fault~~abort occurs.

5. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.

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 0;
Z <- {dst string} EQL 0;
V <- {decimal overflow};
C <- 0;

Exceptions:

reserved operand
decimal overflow

Opcodes:

F9 CVTLP Convert Long to Packed

Description:

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

Notes:

1. After execution:

   R0 = 0

   R1 = 0

   R2 = 0

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

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

3. If all destination digits are zero, Z is set and N is cleared. This is true even if the result overflows.

4. 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 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- 0;
```

Exceptions:

```
reserved operand
integer overflow
```

Opcodes:

```
36  CVTPL  Convert Packed to Long
```

Description:

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

Notes:

1. After execution:

   R0 = 0

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

   R2 = 0

   R3 = 0

2. The destination operand, R0 through R3, and the condition codes are UNPREDICTABLE on a reserved operand fault 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 R0 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.

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 0;
Z <- {src string} EQL 0;
V <- {decimal overflow};
C <- 0;
```

Exceptions:

```
reserved operand
decimal overflow
```

Opcodes:

```
24    CVTPT   Convert Packed to Trailing Numeric
```

Description:

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

Conversion is effected by using the highest addressed byte of the source string (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:

   R0 = 0

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

R2 = 0

R3 = address of the most significant digit of the destination string

2. The destination string, R0 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 fault 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.

5. 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).

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 0;
Z <- {dst string} EQL 0;
V <- {decimal overflow};
C <- 0;
```

Exceptions:

```
reserved operand
decimal overflow
```

Opcodes:

```
 26   CVTTP   Convert Trailing Numeric to Packed
```

Description:

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

Conversion is effected by using the highest addressed (trailing) byte of the source string as an unsigned index into a 256 byte table whose 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 fault occurs if:

    1. The length of the source trailing numeric string is outside the range 0 through 31.

    2. The length of the destination packed decimal string is outside the range 0 through 31.

3. The source string contains an invalid byte. An invalid byte is any value other than ASCII "0" 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:

   R0 = 0

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

   R2 = 0

   R3 = address of the byte containing the most significant digit of the destination string.

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

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

5. If the length of the source string is 0, the destination packed decimal string is set identically equal to 0, 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.

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 0;
Z <- {src string} EQL 0;
V <- {decimal overflow};
C <- 0;

Exceptions:

reserved operand
decimal overflow

Opcodes:

08 CVTPS Convert Packed to Leading Separate Numeric

Description:

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

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

Notes:

1. After execution:

   R0 = 0

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

   R2 = 0

   R3 = address of the sign byte of the destination string

2. The destination string, R0 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 fault 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.

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 0;
Z <- {dst string} EQL 0;
V <- {decimal overflow};
C <- 0;
```

Exceptions:

```
reserved operand
decimal overflow
```

Opcodes:

09	CVTSP	Convert Leading Separate Numeric to Packed

Description:

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

Notes:

1. A reserved operand fault occurs if:

    1. The length of the source Leading Separate numeric string is outside the range 0 through 31.

    2. The length of the destination packed decimal string is outside the range 0 through 31.

    3. The source string contains an invalid byte. An invalid byte is any character other than an ASCII "0" through "9" in a digit byte or an ASCII "+", "<space>", or "-" in the sign byte.

2. After execution:

    R0 = 0

R1 = address of the sign byte of the source string

R2 = 0

R3 = address of the byte containing the most significant digit of the destination string.

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

4. 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.

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 <3:0>*(10 ** (-cnt-1)))
                * (10 ** cnt) ;
```

Condition Codes:

```
N <- (dst string) LSS 0;
Z <- (dst string) EQL 0;
V <- (decimal overflow);
C <- 0;
```

Exceptions:

reserved operand
decimal overflow

Opcodes:

F8	ASHP	Arithmetic Shift and Round Packed

Description:

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

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

Notes:

1. After execution:

   R0 = 0

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

   R2 = 0

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

2. The destination string, R0 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 fault occurs.

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

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

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

6. The round operand is normally five. Truncation may be accomplished by using a zero round operand.

Title: VAX-11 Edit Instruction -- Rev 5

Specification Status: Fully approved

Architectural Status: under ECO control

File: SR4FR5.RNO

PDM #: not used

Date: 2-Nov-78

Superseded Specs: Rev 4

Author: W. Strecker

Typist: B. Call

Reviewer(s): R. Blair, R. Brender, D. Cane, K. Chapman, P. Conklin, D. Cutler, R. Grove, D. Hustvedt, J. Leonard, P. Lipman, M. Payne, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 4 describes the instructions generally used by all software across all implementations of the VAX-11 architecture. For convenience of review and editting, chapter 4 is separated into a number of specifications. This specification contains the edit instruction.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Initial proposal	Blair	Aug-75
Rev 2	Reduce context	Conklin	Feb-76
Rev 3	Complete design	Conklin	May-76
Rev 4	Prune	Conklin	11-Apr-77
Rev 5	Re-issue		2-Nov-78

Rev 4 to Rev 5:

1. No changes

Rev 3 to Rev 4:

1. Drop insert zero/slash and skip zero/slash because infrequent.

2. Combine fixed inserts and insert or protect. Add one byte EO$SET_SIGNIF.

3. Drop move numeric or fill (can use EO$MOVE with significance clear).

4. Combine move into EO$MOVE by setting significance.

5. Drop insert protection register.

6. Drop insert comma/period as optimizations not worth it now.

7. Drop byte repeats. Make repeats be 0 reserved and 1..15 in low nibble.

8. Require EO$END instead of running off the end.

9. Combine protection and fill into one register.

10. Combine currency and sign into one register. This combines the logic of floating currency and sign.

11. Drop optimized combination of EO$BLANK_ZERO and EO$END.

12. Replace options and sign testing fills with EO$LOAD_PLUS and EO$LOAD_MINUS.

    The above pruning results in an average growth in the length of an edit pattern string of about 4 bytes. This is offset by removing two operands (typically 2 to 4 bytes).

13. Add EO$REPLACE_SIGN to fixup -0. This saves about 10 bytes and several operators over the previous proposal.

14. Confine the pattern to associate with the field. Drop destination length operand and adjust destination.

15. Drop input editting--this would be a distinct instruction when needed in volume. Change name to EDITPC.

16. Map flags onto the condition codes.

17. Do not touch R6 and R7.

18. Drop flag set/clear/branch and position add/subtract.

19. Invert sense of significance flag.

20. Assign pattern op codes.

21. Make 0 length of EO$BLANK_ZERO and EO$REPLACE_SIGN be UNPREDICTABLE.

22. Use the term "abort" rather than "fault" when wrong number of input digits.

23. Change "UNDEFINED" to "UNPREDICTABLE".

24. Add usage under notes.

25. Restrict EO$ADJUST_INPUT to 31 digits.

26. Set R0-R1 at end to point to input string.

27. Correct ISP to not increment pattern pointer at end.

28. Add comments to pseudo ISP to aid the initial reader.

Rev 2 to Rev 3:

1. Add option to set protection register to blank vs. asterisk. reason: removes unnecessary duplication of operators.

2. Add operator EOMNF (move numeric or fill). reason: PL/I Y-picture.

3. Add option to force the sign during initialization. reason: handle special sign conventions.

4. Add operator EOMNFS (move numeric or floating sign). reason: handle FORTRAN sign convention.

5. Use R6, R7 to keep extra EDITN context. Reason: keeps left half of counts for normal string contents.

6. Change name to EDITN.

7. Add ability to handle variable length input and output strings.

8. Change to zoned sign and replace EFSS with EFSO.

9. Keep lengths GTRU 0.

10. Add skip numeric to validate numeric string and skip alphabetic for consistency.

11. Support packed format.

12. Limit edit character string to 255 bytes. This drops all word sizes.

13. Drop move numeric or minus--can be done by selecting one of two patterns.

Rev 1 to Rev 2:

1. Combine two instructions into one. Make an initialization option whether or not to examine the input for the presence of a sign. reason: drop operation code.

2. Replace BWZF flag with a pattern operator to protect backward when zero. reason: decrease context; fewer memory cycles.

3. Reduce context to 4 bytes of edit registers, 5 flags, and only one hidden count. reason: reduce context at no loss of functionality.

4. Replace pattern length operand with initialization operand. reason: reduce context.

5. Add specification of handling when input and output lengths do not match.

6. Avoid nibble-sized counts. Replace with byte-sized. reason: consistency with regular instruction set.

7. Reference edit registers by special operators rather than generic mechanics. reason: support of point 3.

8. Replace flag nibble with general set/clear of any flag. reason: generalize and simplify.

9. Add pattern operators to adjust source and destination pointers. reason: generalize and simplify.

10. Set condition codes to reflect the result. reason: consistency with the rest of the architecture.

11. Add input conversion operators. reason: completeness. They are optionally subsettable.

12. On numeric moves and edits, if the input is not a blank or digit, set conversion error. If non-zero, clear Zero flag. reason: correct setting of condition codes.

13. If input minus sign and not output, then set conversion error. reason: correct setting of condition codes.

14. Add specification of behavior on reserved pattern operator. reason: allow simulation of missing operators.

15. Use fill or protection registers for all blanking. reason: flexability.

[End of SR4FR5.RNO]

CHAPTER 4

INSTRUCTIONS

2-Nov-78 -- Rev 5

****
*
* THROW THIS PAGE AWAY.
*
* This is the sixth part of Chapter 4.
*
****

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 non-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 -0 input). The other condition codes are computed and updated as the instruction proceeds. When the EDITPC instruction terminates, registers R0-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};
PSW<V,C> <- 0;
PSW<Z> <- 1;
PSW<N> <- {src has minus sign};
R0 <- srclen;
tmp1 <- 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 <- tmp1;             !length of source string
R1 <- R1 - {tmp1/2}      !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 fault occurs with FPD cleared if srclen GTRU 31. See Chapter 6 for a description of reserved operand faults and FPD.

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

7. The contents of the destination string and the memory preceding it are UNPREDICTABLE if the length covered by EO$BLANK_ZERO or EO$REPLACE_SIGN is 0 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 R0 = -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.

10. On an unimplemented or reserved pattern operator, a reserved operand fault is taken with R3 = location of the faulting pattern operator. The condition codes and other registers are as specified in note 11. This fault 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 10, FPD is set and the condition codes and registers are as follows:

    N = {src has minus sign}

    Z = all source digits 0 so far

    V = non-zero digits lost

    C = significance

    R0 = -zeros<15:0> ' srclen<15:0>

    R1 = current source location

    R2 = ??? ' sign ' fill

R3 = 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

\ The following "picture" editting is outside the scope of EDITPC:

A,X	use MOVC and MOVB
PL/I:E,K	floating point. separate into two integers
PL/I:I,R,T	overpunch. treat as 9 and fixup afterwards
V,P,PL/I:F	scaling by ASHP to get correct position first
BASIC:%	special case code
BASIC:C,L,R,E	MOVC with special code
FORTRAN:*	special code triggered by overflow
FORTRAN:leading -	extra byte in destination string

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 -0
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)		
00	EO$END	
01	EO$END_FLOAT	
02	EO$CLEAR_SIGNIF	
03	EO$SET_SIGNIF	
04	EO$STORE_SIGN	
05..1F	Reserved to DEC	
20..3F	Reserved for all time	
40	EO$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$ADJUST_INPUT	/
48..5F	Reserved to DEC	
60..7F	Reserved to CSS, customers	
80,90,A0	Reserved to DEC	
81..8F	EO$FILL	\
91..9F	EO$MOVE	¦-- repeat count is <3:0>
A1..AF	EO$FLOAT	/
B0..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 0 through 9
        if R0 LEQ 0
        then
                begin
                if R0 EQL 0 then {reserved operand};
                READ <- 0;
                R0<31:16> <- R0<31:16> + 1;      !see EO$ADJUST_INPUT
                end;
        else
                begin
                READ <- (R1)<3+4*R0<0>:4*R0<0>>; !get next nibble
                                                 !alternating high then low
                R0 <- R0 - 1;
                if R0<0> EQL 1 then R1 <- R1 + 1;
                end;
        return;


STORE(char):
        (R5) <- char;
        R5 <- R5 + 1;
        return;
```

Also the following definitions are used:

```
        fill =  R2<7:0>

        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:

04	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 0 then
                begin
                PSW<Z> <- 0;
                PSW<C> <- 1;    !set significance
                end;
        if PSW<C> EQL 0 then STORE(fill)
                else STORE("0" + 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 0 then
                begin
                if PSW<C> EQL 0 then STORE(sign);
                PSW<Z> <- 0;
                PSW<C> <- 1;    !set significance
                end;
        if PSW<C> EQL 0 then STORE(fill)
                else STORE("0" + 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.

5. 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 0 then
        begin
        STORE(sign);
        PSW<C> <- 1;     !set significance
        end;
```

Pattern operators:

01	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 Minus Zero

Purpose:

Fixup the destination sign when the value is minus zero

Format:

pattern len

Operation:

```
if len EQLU 0 then {UNPREDICTABLE};
if PSW<Z> EQL 1 and PSW<N> EQL 1 then
        (R5 - len) <- fill;
```

Pattern operators:

46 EO$REPLACE_SIGN Replace Sign When Minus Zero

Description:

The pattern operator is followed by an unsigned byte integer length. If the value of the source string is minus zero (i.e., if both N and Z are set), then the contents of the fill register is stored into the byte of the destination string length 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 0 or len GTRU 31 then {UNPREDICTABLE};
if R0<15:0> GTRU len
then
        begin
        R0<31:16> <- 0
        repeat R0<15:0> - len do
                if READ NEQU 0 then
                        begin
                        PSW<Z> <- 0;
                        PSW<C> <- 1;    !set significance
                        PSW<V> <- 1;
                        end;
        end;
else R0<31:16> <- R0<15:0> - 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 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 R0<31:16>.

Notes:

If length is not in the range 1 through 31 the destination string, condition codes, and R0 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
                            !described under EDITPC instruction
```

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.

CHAPTER 4

INSTRUCTIONS

26-Oct-78 -- Rev 5

****
*
* THROW THIS PAGE AWAY.
*
* This is the seventh part of Chapter 4.
*
****

Title: Other VAX-11 Instructions -- Rev 5

Specification Status: Fully approved

Architectural Status: under ECO control

File: SR4GR5.RNO

PDM #: not used

Date: 26-Oct-78

Superseded Specs:

Author: D. Bhandarkar

Abstract: Chapter 4 describes the instructions generally used by all software across all implementations of the VAX-11 architecture. For convenience of review and editting, chapter 4 is separated into a number of specifications. This section contains a list of instructions specified in other chapters of this document.

Revision History:

Rev #	Description	Author	Revised Date
Rev 5	Initial distribution	Bhandarkar	26-Oct-78

4.13 OTHER VAX-11 INSTRUCTIONS

The following instructions are specified in other chapters of this document as indicated below.

Opcode	Mnemonic	Instruction	Chapter
0C	PROBER	Probe Read Accessibility	5
0D	PROBEW	Probe Write Accessibility	5
02	REI	Return from Exception or Interrupt	6
BC	CHMK	Change Mode to Kernel	6
BD	CHME	Change Mode to Executive	6
BE	CHMS	Change Mode to Supervisor	6
BF	CHMU	Change Mode to User	6
06	LDPCTX	Load Process Context	7
07	SVPCTX	Save Process Context	7
DA	MTPR	Move To Processor Register	9
DB	MFPR	Move From Processor Register	9

[End of Chapter 4]

Title: VAX-11 Memory Management -- Rev 4

Specification Status:

Architectural Status: under ECO control

File: SR5R4.V09

PDM #: not used

Date: 30-Jul-78

Superseded Specs:

Author: P. Lipman

Typist: J. Bess/D. Lindorfer

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 5 describes the memory mapping and memory protection mechanisms of the VAX-11 series. The areas covered include the description of the virtual address space, the translation of virtual address to physical address, the protection of pages according to access mode, and the PROBE instruction.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	Hastings	Oct-75
Rev 2	Per ECO's 1-18	Hastings	Mar-76
Rev 3	Redesign-Per April Task Force	Lipman	4-Jun-76
Rev 4	Document Registers	Conklin/Taylor	30-Jul-78

Rev 3 to Rev 4:

1. Typos.
2. PTE<24:23> now for software use; <25> and <22:21> are untested MBZ.
3. Include boot initial state.
4. Correct picture formats.
5. Add descriptions of translation buffer, unmapped, MAPEN, TBIA, and TBIS.
6. Add picture of stack on fault.
7. Add example of translation.
8. Cleanup the description of PROBE.
9. P0LR<26:24> and P1LR<31> ignored on MTPR.
10. PROBE uses only mode<1:0>; ignores rest of operand.
11. Remove redundant, confusing pseudo flows.
12. Remove some argumentative justifications.
13. Specify which registers are read/write, etc.
14. SLR is checked on process page table references.
15. Clarify address on fault.
16. M not necessarily maintained in SPT of process page tables.
17. Add software mnemonics.
18. SWU bits renamed to OWN and reserved for owning access mode.
19. M update is not interlocked in multiprocessor.
20. Add comments on accesses across page boundaries.

Rev 2 to Rev 3:

1. Eliminate the concept of segments, moving protection to the page level
2. Streamline the virtual to physical address translation by eliminating references to the PCB and Internal Segment Table.

3. Remove access level from virtual address

4. Eliminate Pointer Segments and Stack Segments

5. Add mode operand to PROBE

6. Assign codes to protection field

7. Change sense of condition codes on PROBE

8. PROBE length is unsigned

9. Protection field is valid even if V=0.

10. P1BR specifies lowest (unused) address in P1-Space

11. Modify bit moved to page table entry

12. Memory Mapping Enable Bit

Rev 1 to Rev 2:

1. Remove execute protection

2. Added flows for translation

[End of SR5R4.RNO]

CHAPTER 5

MEMORY MANAGEMENT

30-Jul-78 -- Rev 4

5.1 INTRODUCTION

This chapter describes the memory mapping and memory protection mechanisms of the VAX-11 series. The VAX-11 memory management has the following goals:

1. Provide a large address space for instructions and data.

2. Permit most software to be run on all implementations across a large range of hardware cost.

3. Allow data structures up to one gigabyte.

4. Provide convenient and efficient sharing of instructions and data.

5. Contribute to software reliability.

A virtual memory system is used to provide a large address space, while allowing programs to run on small memory size hardware configurations. Programs are executed in an execution environment termed a process. The software operating system uses the mechanisms described in this chapter to provide each process with a 4 billion byte address space.

The virtual address space is divided into two equal size address spaces, the process address space and the system address space. The system address space is the same for all processes. The operating system is in the system address space and is written as callable procedures. Thus all system code is available to all other system and user code using a simple CALL. The process address space is separate for each process. However, several processes may have access to the same page, thus providing controlled sharing.

To improve software reliability 4 hierarchical layers of memory access privilege are provided by the access mode mechanism. Protection is specified at the individual page level. For each of the four access levels, a page may be inaccessible, read-only, or read-write. Any location accessible to a lesser privileged mode is also accessible to all more privileged modes. Furthermore for each access mode, any location that is writable is also readable.

5.2 VIRTUAL ADDRESS SPACE

The address space seen by the programmer is a linear array of 4,294,967,296 bytes. A virtual address is a 32 bit unsigned integer specifying a byte location in the address space.

This virtual address space is too large to be contained in any presently available main memory. Hence a means for mapping the active part of the virtual address space to the available physical address space is required. Also, protection between processes is required. The operating system controls the memory management tables that map virtual addresses into physical memory addresses. The inactive but used parts of the virtual address space are mapped by the operating system onto external storage media. The virtual address space is broken into 512 byte units termed pages. The page is the unit of relocation and protection. See section 5.4 for a description of the address translation process.

The virtual address space is split into two parts. The lower half is distinct for each process running on the system, while the upper half is shared by all processes.

5.2.1 Process Space

The lower half is termed "process space". Each process has a separate address translation map for per process space, so the per process spaces of all processes are completely disjoint (see 5.8.3 for controlled sharing in per process space). The address map for per process space is context switched when the process running on the system is changed (see chapter 7).

Process space is split by bit 30 into two regions termed the P0 and P1 regions of the process virtual address space. See section 5.4.3.

5.2.2 System Space

The upper half 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.

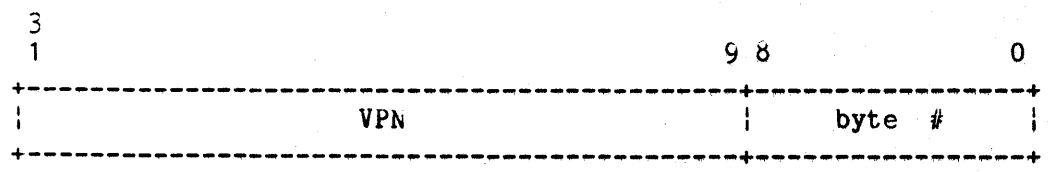
5.2.3 Page Protection

Independent of its location in the virtual address space a page may be protected according to its use. Thus 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.

5.2.4 Virtual Address

In order to reference each instruction and operand in memory the processor generates a 32-bit virtual address that has the following format:



VPN <31:9> The Virtual Page Number field specifies the virtual page to be referenced. There are 2**23 (i.e. 8,388,608) pages in the virtual address space.

Byte # <8:0> The byte number field specifies the byte address within the page. A page contains 512 bytes.

5.2.5 Virtual Address Space Layout

Access to each of the three regions (P0, P1, System) is controlled by a length register (P0LR, P1LR, SLR) (see section 5.3.3). Within the limits set by the length registers, the access is controlled by a page table that specifies the validity, access requirements, and location of each page in the region.

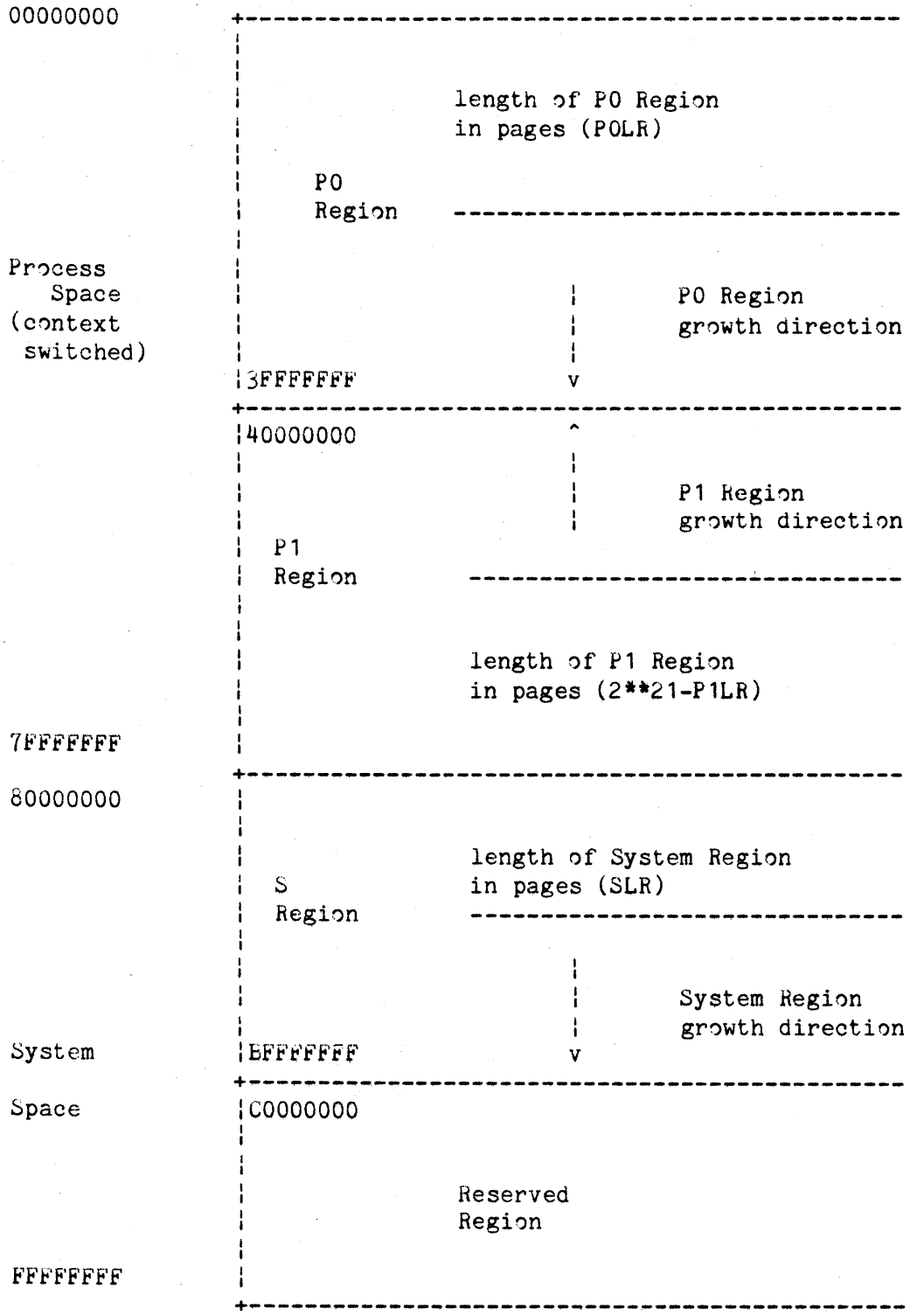


Figure 5-1
Virtual Address Space

5.3 ACCESS CONTROL

Access control is the function of validating whether a particular type of memory access is to be allowed to a particular page. Every page has associated with it a protection code that specifies for each mode whether or not read or write references are allowed. Additionally, each address is checked to make certain that it lies within the P0, P1, or system region.

5.3.1 Mode

There are 4 hierarchically ordered modes in the processor. The modes in the order of most privileged to least are:

0 - 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 mode at which the processor is currently running is stored in the Current Mode field of the Processor Status Longword (PSL) (see Chapter 6).

5.3.2 Protection Code

Associated with each page is a protection code that describes the accessibility of the page for each mode. The protection codes available allow choice of protection for each access level 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.

This results in 15 possibilities. The protection code is encoded in a 4 bit field in the Page Table Entry (see 5.4.1) as follows:

CODE DECIMAL	CODE BINARY	MNEMONIC	K	E	S	U	COMMENT
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	

- \- - no access
- R - read only
- RW - read write

- K - Kernel
- E - Executive
- S - Supervisor
- U - User

Software symbols are defined using PTE$K_ as a prefix to the above mnemonics.

This code was chosen to keep the complexity of hardware access checking reasonable 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 mode
    RM is left 2 bits of code
    WM is one's complement of right 2 bits of code
```

5.3.3 Length Violation

Every virtual address is constrained to lie within one of the valid addressing regions (P0, P1, or System). The algorithm for making these checks is a simple limit check. The formal notation for this check is:

```
case VAddr<31:30>
    set
    [0]:                                    !P0 region
        if ZEXT( VAddr<29:9> ) GEQU P0LR
            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;
```

5.3.4 Access Control Violation Fault

An access control fault occurs if the current mode of the PSL and the protection field(s) for the page(s) about to be accessed indicate that the access would be illegal (see 5.6). A fault of this type will also occur if the address causes a length violation to occur.

5.4 ADDRESS TRANSLATION

The action of translating a virtual address to a physical address is governed by the setting of the Memory Mapping Enable (MME) bit (see section 5.5.1). When MME is 0 the low order bits of the virtual address are the physical address and there is no page protection. The number of bits is implementation dependent. This section describes the address translation process when MME is 1.

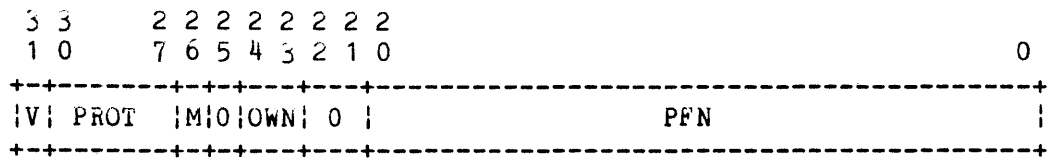
The address translation routine is presented with a virtual address, an intended access (read or write) and a mode against which to check that access. If the access is allowed and the address maps without faulting, the output of this routine is the physical address corresponding to the specified virtual address.

The mode that is used is normally the Current Mode field of the PSL but per process page table entry references use Kernel mode

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, however, the operation to be performed is a modify (i.e. read followed by write) the intended access for the read portion is specified as a write.

5.4.1 Page Table Entry (PTE)

All virtual addresses are translated to physical addresses by means of a Page Table Entry (PTE) that has the following format.



Valid <31> valid bit - governs the validity of the M bit and PFN field. V=1 for valid, V=0 for not valid.

PROT <30:27> protection field - this field is always valid and is used by the hardware even when V=0 (see 5.3.2).

Modify <26> modify bit - set if page has already been recorded as modified. M=0 if page has not been recorded as modified. Used by hardware only if V=1. Hardware sets this bit on a valid, access allowed memory access associated with a modify or write access, and optionally on a PROBEW or implied probe-write. If a write or modify reference crosses a page boundary and one page faults, it is UNPREDICTABLE whether PTE<M> for the other page is set before the fault.

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> reserved for software use as the access mode of the owner of the page (i.e., the mode allowed to alter the page); not examined or altered by hardware.

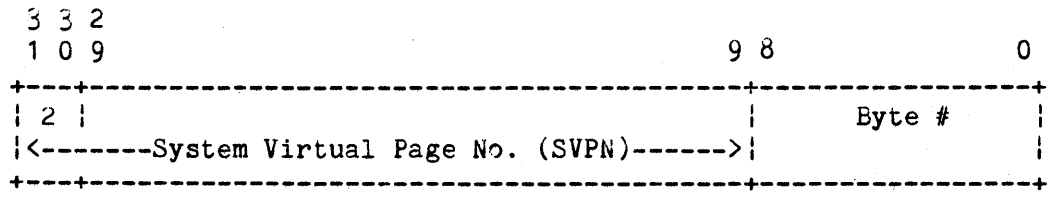
PFN <20:0> Page Frame Number - the upper 21 bits of the physical address of the base of the page. Used by hardware only if V=1.

0	Bits 25 and 22:21 are reserved to DIGITAL and must be zero. The hardware does not necessarily test that these are zero because the PTE is established only by privileged software.

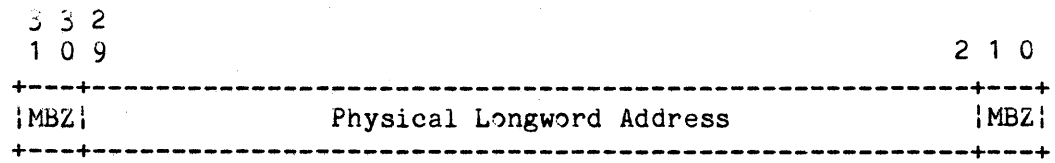
Software symbols are defined for the above fields using PTE$ as the prefix.

5.4.2 System Space Address Translation

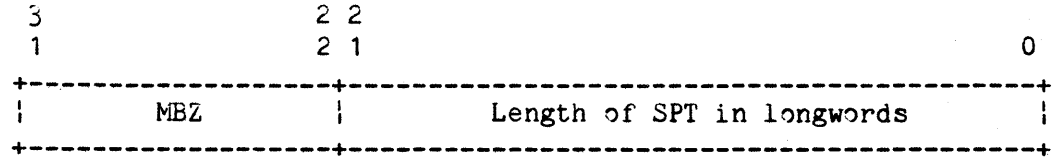
A virtual address with <31:30>=2 is an address in the system virtual address space.



The system virtual address space is defined by the System Page Table (SPT), which is a vector of Page Table Entries (PTE's). The physical base address of the SPT is contained in the System Base Register (SBR). The size of the SPT in longwords, i.e., the number of PTE'S, is contained in the System Length Register (SLR). The PTE addressed by the SBR maps the first page of System Space, i.e., virtual byte address 80000000(hex).



(read/write)
System Base Register (SBR)



(read/write)
System Length Register (SLR)

The virtual page number is bits <31:9> of the virtual address. However, system virtual addresses have VAddr<31:30>=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 section 5.8).) A 22 bit length field is required to express the values 0 through 2**21 inclusive. At bootstrap time, the contents of both registers is UNPREDICTABLE.

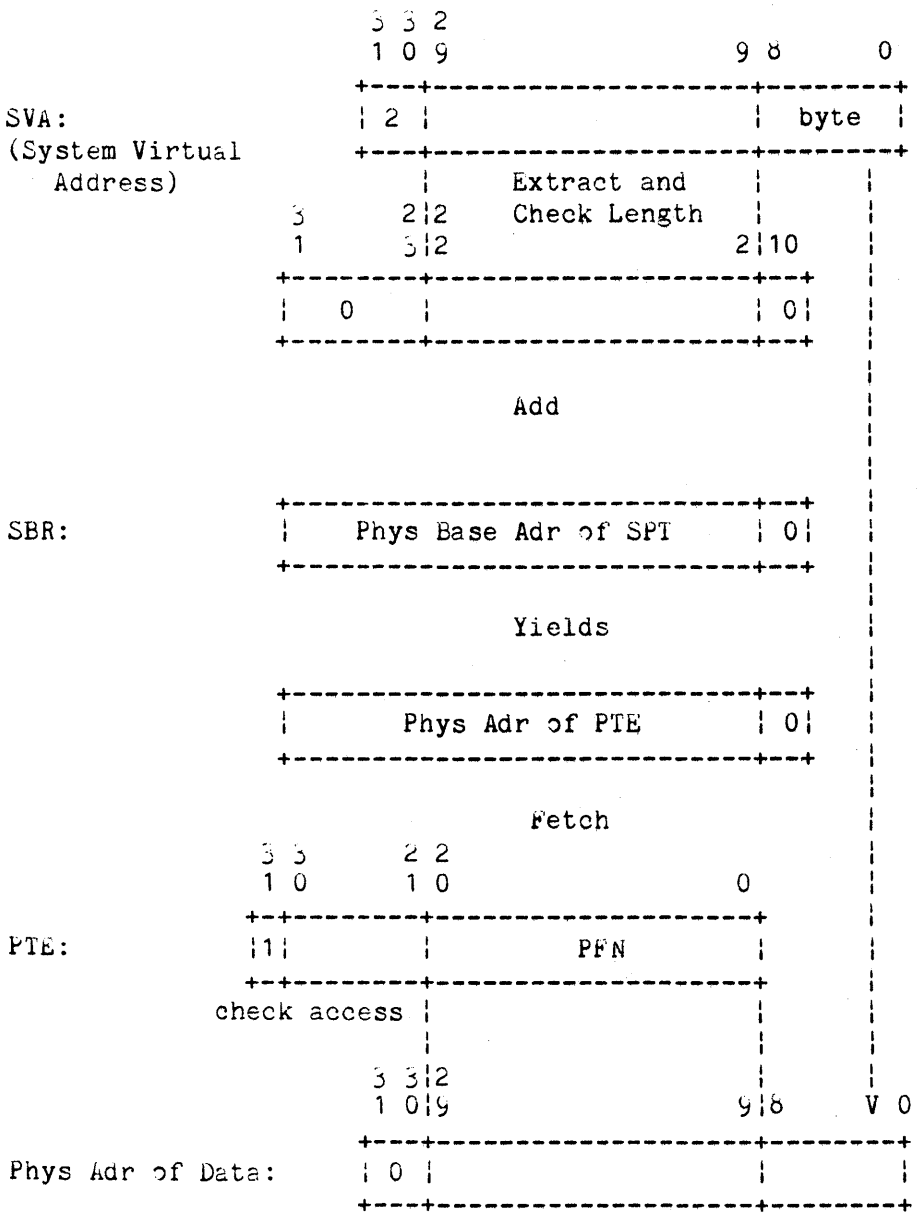


Figure 5-2a
System Virtual to Physical Translation

Thus, the arithmetic necessary to generate a physical address from a system region virtual address is:

```
SYS_PA = (SBR+4*SVA<29:9>)<20:0>'SVA<8:0>      !System Region
```

5.4.3 Process Space Address Translation

The process virtual address space is split into two separately mapped regions according to the setting of bit 30 in the process virtual address. If bit 30 is 0, the P0 region of the address space is selected and if bit 30 is 1, the P1 region is selected.

The P0 region of the address space 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. In contrast, the P1 region of the address space 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.

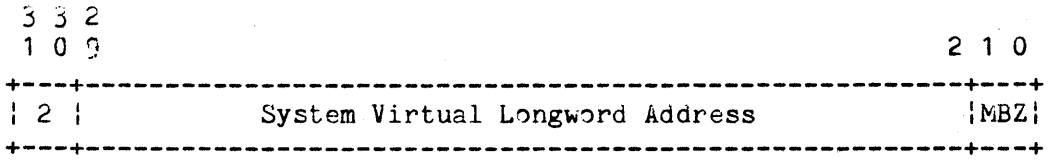
Each region (P0 and P1) of the process virtual space is described by a virtually contiguous vector of Page Table Entries. In contrast with the System Page Table which is addressed with a physical address, these two page tables are addressed with virtual addresses in the system region of the virtual address space. Thus for Process Space the address of the PTE is a virtual address in System Space and the fetch of the PTE is simply a fetch of a longword using a system virtual address.

There is a particularly compelling 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 PTE's (i.e., 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.

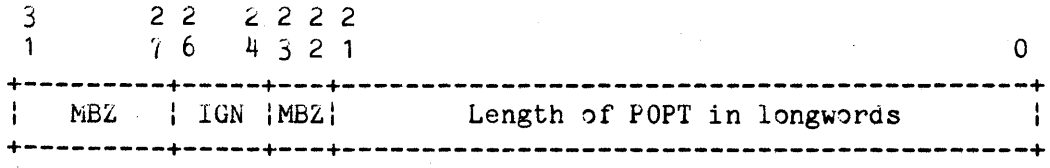
A process space translation that causes a translation buffer miss will cause one memory reference for the process PTE associated with this translation (see 5.5.2). If the virtual address of the page containing the process PTE is also missing from the translation buffer, a second memory reference is required to fetch the system PTE for the system virtual address of the process PTE. 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" (i.e., 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 section 5.6).

5.4.4 P0 Space

The P0 region of the address space is mapped by the P0 Page Table (P0PT) that is defined by the P0 Base Register (P0BR) and the P0 Length Register (P0LR). P0BR contains a virtual address in the system half of the virtual address space which is the base address of P0PT. P0LR contain the size of P0PT in longwords, i.e., the number of page table entries. The PTE addressed by P0BR maps the first page of the P0 region of the virtual address space, i.e., virtual byte address 0.



(read/write)
P0 Base Register (P0BR)



(read/write)
P0 Length Register (P0LR)

The virtual page number is bits <29:9> of the virtual address. Thus, there could be as many as 2**21 pages in the P0 region. A 22 bit length field is required to express the values 0 through 2**21 inclusive. P0LR<26:24>are ignored on MTPR and reads back 0 on MFPR. At bootstrap time, the contents of both registers is UNPREDICTABLE. An attempt to load P0BR with a value less than 2**31 or greater than 2**31+2**30-1 results in a reserved operand fault in some implementations.

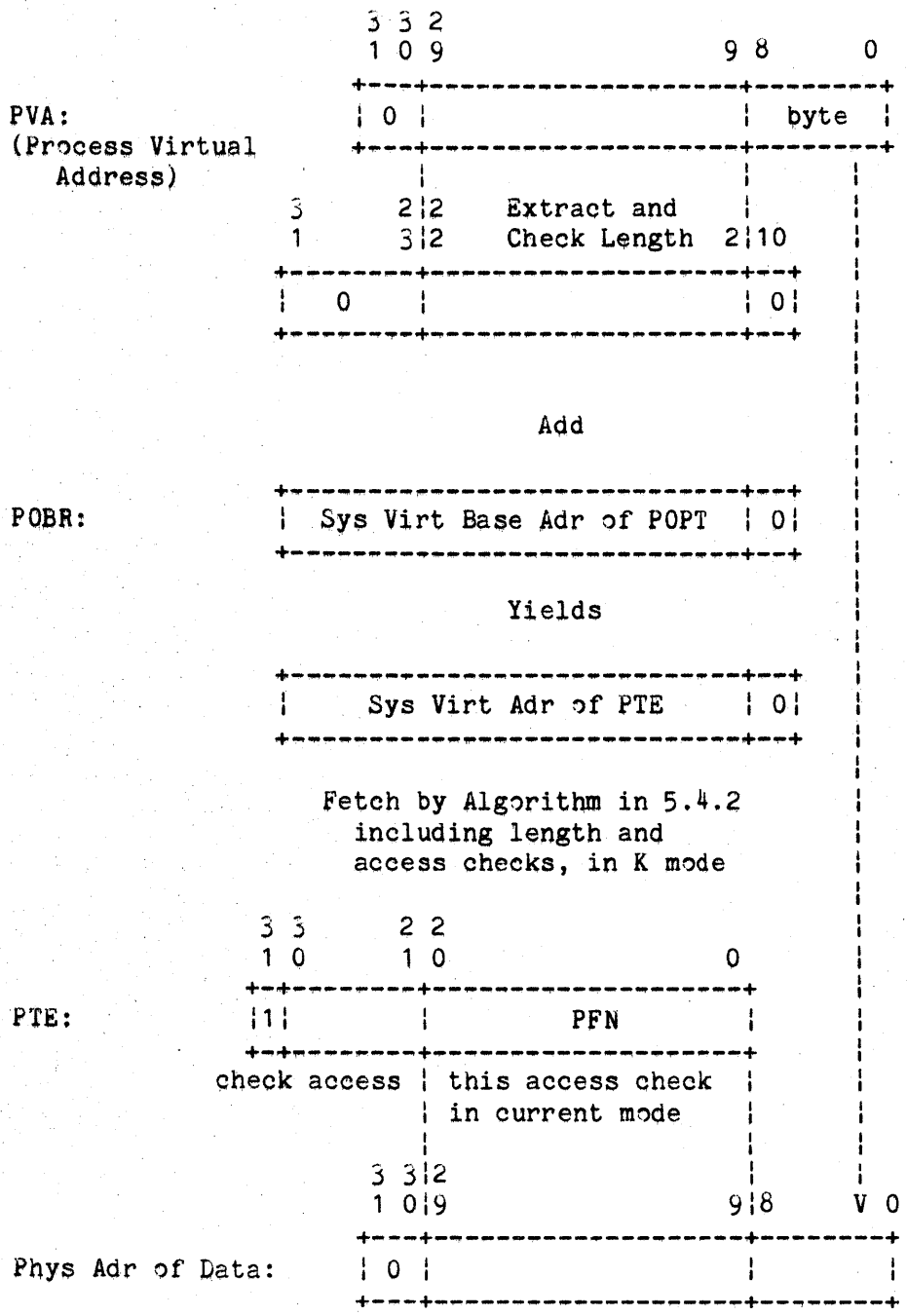


Figure 5-2b
P0 Virtual to Physical Translation

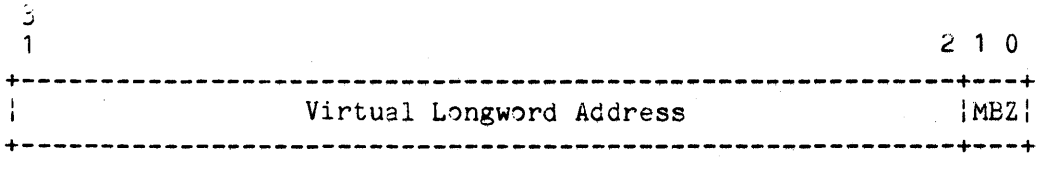
Thus, the arithmetic necessary to generate a physical address from a P0 region virtual address is:

```
PVA_PTE = P0BR+4*PVA<29:9>                          !P0 Region
PTE_PA  = (SBR+4*PVA_PTE<29:9>)<20:0>'PVA_PTE<8:0>
PROC_PA = (PTE_PA)<20:0>'PVA<8:0>
```

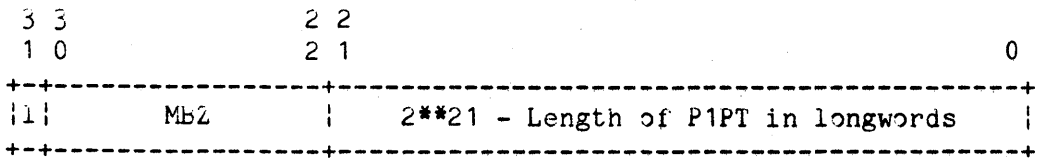
5.4.5 P1 Space

The P1 region of the address space is mapped by the P1 Page Table (P1PT) that is defined by the P1 Base Register (P1BR) and the P1 Length Register (P1LR). Because P1 space grows backwards and because a consistent hardware interpretation of the base and length registers was desired, P1BR and P1LR describe the portion of P1 Space that is not accessible. P1BR contains a virtual address of what would be the PTE for the first page of P1, i.e., virtual byte address 40000000(hex). P1LR contains the number of non-existent PTE's.

Note that the address in P1BR is not necessarily an address in System Space, but all the addresses of PTEs must be in System Space.



(read/write)
P1 Base Register (P1BR)



(read/write)
P1 Length Register (P1LR)

P1LR<31> is ignored on MTPR and reads back 0 on MFPR. At bootstrap time, the contents of both registers is UNPREDICTABLE. An attempt to load P1BR with a value less than 2**31-2**23 (7F800000, hex) or greater than 2**31+2**30-2**23-1 results in a reserved operand fault in some implementations.

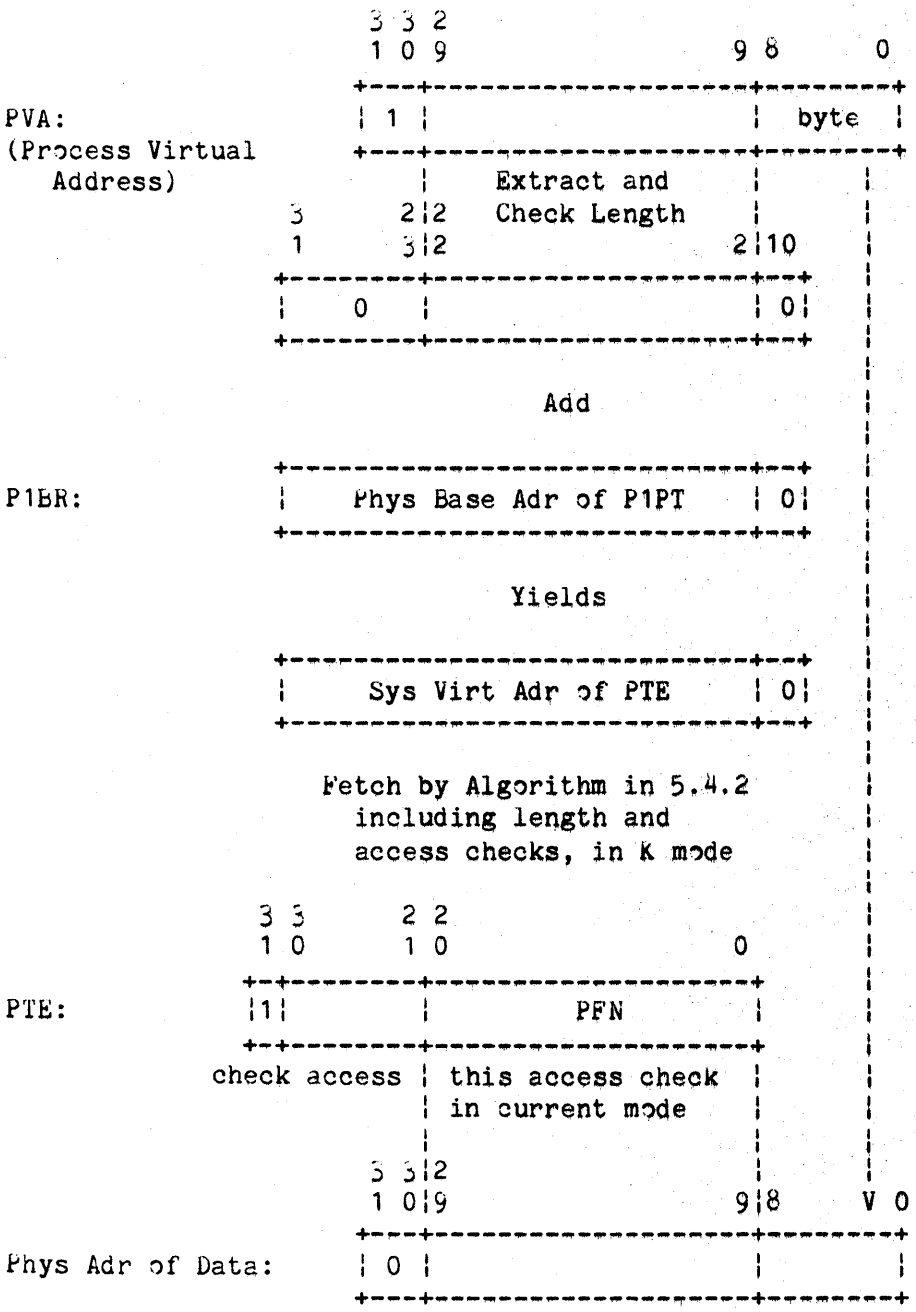


Figure 5-2c
P1 Virtual to Physical Translation

Thus, the arithmetic necessary to generate a physical address from a P1 region virtual address is:

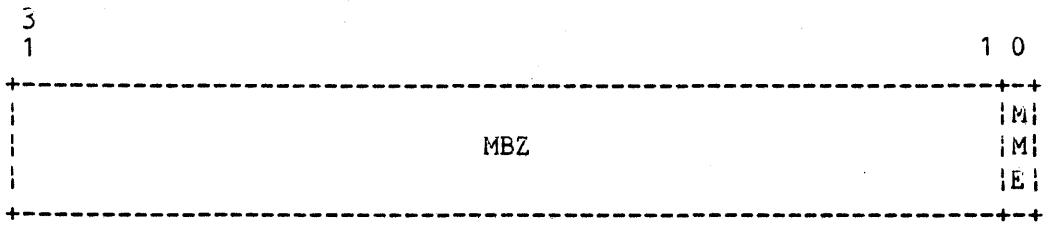
```
PVA_PTE = P1BR+4*PVA<29:9>                          !P1 Region
PTE_PA  = (SBR+4*PVA_PTE<29:9>)<20:0>'PVA_PTE<8:0>
PROC_PA = (PTE_PA)<20:0>'PVA<8:0>
```

5.5 MEMORY MANAGEMENT CONTROL

There are three additional privileged registers used to control the memory management hardware. One register is used to enable and disable memory management, the other two are used to clear the hardware's address translation buffer when a page table entry is changed.

5.5.1 Memory Management Enable

The privileged register MAPEN contains the value of 0 or 1 according to whether the memory management described in the rest of this chapter is disabled or enabled respectively.



(read/write)
Map Enable Register (MAPEN)

At bootstrap time, this register is initialized to 0.

When memory management is disabled, virtual addresses are mapped to physical addresses by ignoring their high order bits. All accesses are allowed in all modes and no modify bit is maintained.

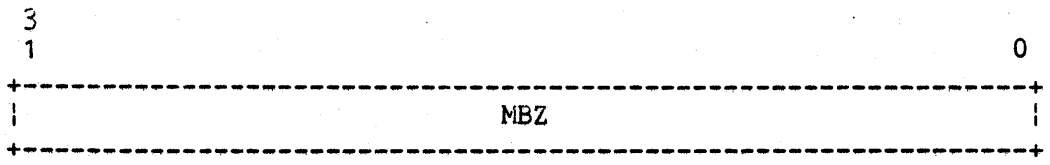
5.5.2 Translation Buffer

In order to save actual memory references when repeatedly referencing 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.

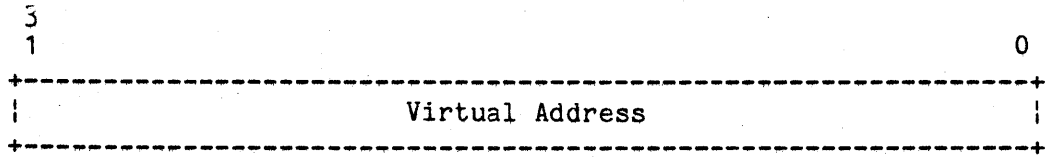
Whenever the process context is loaded with LDPCTX the translation buffer is automatically updated (i.e., the process virtual address translations are invalidated). However, whenever a page table entry for the system or current process regions is changed other than to set PTE<V> the software must notify the translation buffer of this by moving a virtual address within the corresponding page into TBIS.

Whenever the location or size of the system map is changed (SBR, SLR) the entire translation buffer must be cleared by moving 0 into TBIA.

Since the contents of the translation buffer at bootstrap time is UNPREDICTABLE, the entire translation buffer must be cleared by moving 0 into TBIA before enabling memory management.



(write only)
Translation Buffer Invalidate All (TBIA)

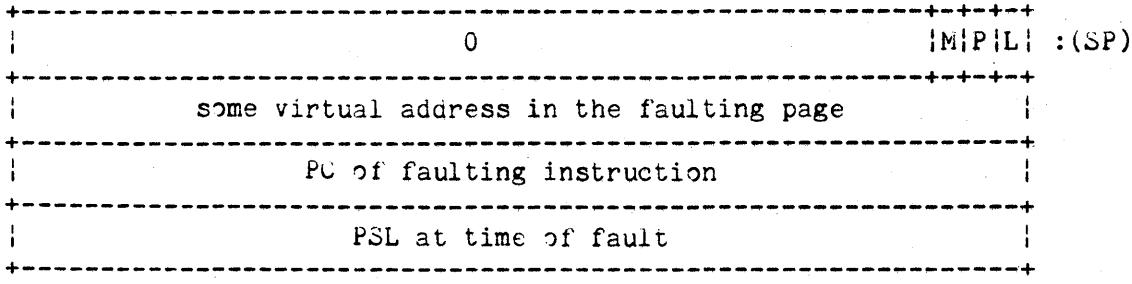


(write only)
Translation Buffer Invalidate Single (TBIS)

5.6 FAULTS AND PARAMETERS

There are two types of faults associated with memory mapping and protection (see Chapter 6 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 access to the page for the specified mode would be illegal. Note that these two faults have distinct vectors in the System Control Block. If both Access Control Violation and Translation Not Valid 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 redo the length check a "length violation" indication is stored in the fault parameter word described below.



The same parameters are stored for both types of fault. The first parameter pushed on the Kernel stack after the PSL and PC is some virtual address in the same page as the initial 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 bit is stored in the fault parameter word indicating that the fault occurred in the PTE reference.

The second parameter pushed on the Kernel stack contains the following information:

L <0> 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 0 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 0 if the program's intended access was a read.

5.7 PRIVILEGED SERVICES AND ARGUMENT VALIDATION

5.7.1 Changing Modes

There are four instructions provided to allow a program to change the mode at which it is running to a more privileged mode and transfer control to a service dispatcher for the new mode. These instructions,

CHMK	change mode to Kernel
CHME	change mode to Exec
CHMS	change mode to Super
CHMU	change mode to User

that are described in detail in Chapter 6, provide the only 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.

5.7.2 Validating Address Arguments (PROBE Instructions)

Two instructions, PROBER and PROBEW, are provided to allow privileged services to check addresses passed as parameters. To avoid protection holes in the system a service routine must always validate that its less privileged caller could have directly referenced the addresses passed as parameters (see Appendix J).

PROBE Probe accessibility

Format:

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

Operation:

```
prob_mode <- MAXU (mode<1:0>, PSL<PRV_MOD>)
condition codes <- {accessibility of base} and
                   {accessibility of {base+ZEXT(len)-1}}
                   using probe_mode
```

Condition Codes:

```
N <- 0;
Z <- if {both accessible} then 0 else 1;
V <- 0;
C <- C;
```

Exceptions:

translation not valid

Opcodes:

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

Description:

The PROBE instruction checks the read or write accessibility of the first and last byte specified by the base address and the zero extended length. The protection is checked against the mode specified in the mode operand which is restricted (by maximization) from being more privileged than the Previous Mode field of the PSL. Note that probing with a mode operand of 0 is equivalent to probing the mode specified in PSL <PRV_MOD>.

The following flows describe the operation of PROBE on each of the virtual addresses it is checking. Note that probing an address only returns the accessibility of the page(s) and has no affect on their residency.

1. Lookup 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 5.5.2 and 5.5.3 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 (see 5.5.2), 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. Lookup 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 (see 5.5.2). If length violation, then return No Access and EXIT. \This length violation is clearly an operating system error and should never happen. Perhaps it should fault.\

    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 saves a page full of no access, not valid PTE's.

    5. If the valid bit in T1 is 0, 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 5.5.2), fetch the PTE, use the protection field to determine the accessibility, and EXIT.

5.7.3 Notes On The PROBE Instructions

1. The valid bit of the page table entry mapping the probed address is ignored.

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

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

4. 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.

5. It is UNPREDICTABLE whether the Modify bit of the examined page table entry is set by a PROBEW.

5.8 ISSUES

5.8.1 Physically Contiguous System Page Table

During the design the issue was raised that a physically based physically contiguous SPT might require a large amount of memory to handle a reasonable number of very big processes.

5.8.1.1 Size Of SPT - To examine the size of SPT, note first that 1 page of SPT maps 64 KB of system virtual address space. What is mapped in the system virtual space?

1. Operating System code and data (excluding memory management data) - 64 KB to 96 KB, 1 to 1.5 pages.

2. Memory Management Data for physical page management - 4 to 6 longwords per physical page of memory. One longword of page table maps 1 page of memory management data which handles 24 physical pages of memory. 1 page of page table handles 3K physical pages = 1.5 MB of physical memory

3. Shared Code -

    1. command interpreter

    2. debugger

3. record manager

4. OTS's FORTRAN, COBOL, BASIC

Allowing 16 KB for each of the above items, the total is 96 KB or 1 1/2 pages of SPT.

4. Process Page Tables

One longword of SPT maps one page of process page table which in turn maps 64 KB of process virtual address space. Sixteen longwords of SPT maps 1 MB of process virtual address space. One page of SPT maps 8 MB. A very straight forward balance set management design that reserved a fixed (SYSGEN) number of balance set slots each with a fixed (also SYSGEN) maximum virtual address space would use only 2 pages of SPT to allow 16 processes of up to 1 MB each in the balance set.

It would appear from the foregoing analysis that a 6 page SPT would handle a very reasonable system and increasing the 1 MB process virtual space to 4 MB and 16 processes in the balance set would add only 6 more pages of SPT for a total of 12. A smaller system with 256 KB of memory and 8 balance set processes each 512 KB maximum size would need about 3 pages of SPT.

5.8.2 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.

5.8.3 Sharing

To discuss sharing, it is useful to assume the concept of a section in the operating system. A section is a collection of pages that have some relationship to each other. Though units as small as pages may indeed be shared, sections are the usual unit of sharing.

5.8.3.1 Shared Section In Process Space - Sharing in the process half of the virtual address space requires that the page table fragments for the sections being shared be replicated in the process page table(s). Clearly this introduces multiple PTE's for the same physical page. This is a problem traditionally avoided by one or more levels of indirection, i.e., the PTE points to the shared PTE that points to the page. We can avoid introducing this level of indirection in the hardware by observing the following software rules:

1. A share count is maintained for each shared page in memory and in effect counts the number of direct pointers to that page.

2. When a process releases a page from its working set and it is a shared page as indicated in the working set data base, the private PTE must be changed to point at the shared PTE for the page, and the private copy of the modify bit must be OR'ed into the shared PTE. Then the share count is decremented and if the count is now 0, the page is released and the shared PTE is updated to reflect that. Note that the process' working set data base allows it to find its private PTE and the physical page data base points to the shared PTE.

3. When a process gets an invalid page fault one of the possible states of the "invalid" PTE is that it points to a shared PTE. Of course that PTE might say that the page was not resident requiring a page read. Whether or not the read was necessary, the shared PTE is eventually copied to the private PTE and the share count of the page is incremented.

4. Note that throwing a process out of the balance set is the equivalent of releasing all its pages (see 2.)

5. The use of the indirect page pointer as a software only mechanism seems to be adequate for this form of sharing. It should be noted that it is very difficult to change the PFN of a page in memory when it is actively being shared. That would require a scan of the page tables for all the processes in the balance set.

5.8.3.2 Shared Sections In System Space - When a process is using a shared section in the system region of the address space, it is referencing a single shared page table. Since it is possible for a process to simply reference such a shared section without ever having declared its intention to do that, the operating system must be prepared to do something reasonable when such a reference faults. A straight forward design for this kind of sharing is:

1. Have programs explicitly declare their intention to use each shared system section. This could be done statically at compile or link time or dynamically at runtime.

2. Have the balance set manager swap in and lock down the entire section when the process intending to use it is swapped in.

3. Of course the balance set manager maintains share counts on the section and only discards its pages when no process in the balance set wants it.

4. If a process faults such a page because it failed to declare its intention to use the section, then that is considered a programming error.

Another approach for shared system sections allows a process to reference pages of the section with no prior declaration of its intent to use them. Such pages would be demand paged within a pool of pages reserved for that purpose. There would be a list of the pages in use in that pool and a fault for a new one would cause one in the pool to be replaced. This would use the same sort of working set management that is used for the process address space but it would be global across processes.

5.8.4 Protection Check Before Valid Check

The Page Table Entry has been defined as having a valid bit that only controls the validity of the Modify Bit and Page Frame Number field. The protection field is defined as always being valid and checked first.

The the motivation for this design is the behavior the PROBE instruction would exhibit if the Valid bit had to be set before it could check protection. PROBE would actually have to fault in the page to make it valid so that it could check the protection and then indicate whether or not the intended access was permissable. For the vast percentage of PROBE instructions, the access is permitted and faulting the page in the PROBE is certainly not unreasonable. But a program could be run in user mode that would PROBE all around in the System region of the virtual address space faulting all the swappable pages of the System. Though this would not violate the integrity of the operating system it certainly would mess up any statistics that the system might be gathering about the relative goodness of the swappable pages.

The only drawback to the protection check before valid check design seem to be a slight loss of flexibility in the memory management software. If the valid bit governed the rest of the longword, then the software could use all of the remaining 31 bits when V=0. This is not an obvious winner since if the protection field isn't maintained in the PTE it must be kept somewhere else.

[End of Chapter 5]

Title: VAX-11 Exceptions and Interrupts -- Rev 6

Specification Status:

Architectural Status: under ECO control

File: SR6R6.RNO

PDM#: not used

Date: 31-Jan-79

Superseded Specs: Rev 5

Author: J. Leonard, P. Conklin

Typist: N. Ford, B. Call, J. Bess

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 6 describes the mechanisms by which exception and interrupt conditions are presented to software, and the means by which programs change their access level.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Initial Distribution of SRM	Hastings	Oct-75
Rev 2	April Meeting	Leonard	7-May-76
Rev 3	Approval by STAR Task Force	Leonard	3-Jun-76
Rev 4	ECOs and Reedit	Conklin	20-Apr-77
Rev 5	Editorial	Conklin/Taylor	30-Jul-78
Rev 6	Floating Faults	Bhandarkar	31-Jan-79

Rev 5 to Rev 6:

1. Floating Fault Enable
2. xxxQUE are unaligned
3. ECO IS, IPL for exceptions

Rev 4 to Rev 5:

1. Add SW mnemonics for arithmetic traps.
2. xxxQUE are aligned (ECO).
3. Clean up Initiate Exception or Interrupt.

Rev 3 to Rev 4:

1. Add reserved length on EDIT (EDITPC ECO).
2. Make Kernel Stack not valid IPL be 1F, power fail IPL be 1E (KS not valid ECO).
3. Correct numerous typos including residual -16 and -14.
4. Add POLYx reserved operand. (POLY ECO)
5. Change "disaster" to "urgent".
6. Compatibility mode is a fault except odd address which is an abort. Reconcile codes with chapter 10.
7. Trace is a trap (TP is a fault).
8. IPL 1F is for exception vector<1:0>=1.
9. IPL 1E is for power fail only.
10. If interrupt routine modifies FPD, regs, or CC, then results are UNPREDICTABLE.
11. Add console terminal and interval timer vectors and IPL.
12. Change "STAR device" to "NEXUS".
13. CHMK and REI do not get KS not valid or IS not valid.
14. REI verifies IPL NEQU 0 if IS NEQU 0.
15. Add BIC/BISPSW to reserved operand list.
16. Remove terms ISR, ESR, WCS.

17. Add flow for initiation of exception or interrupt.

18. Add between instruction flow.

19. If FPD=1 saving PSL, then TP cleared. Verify on REI.

20. Verify on REI that if IS=1, then current mode=0.

21. Correct PROBEW in CHMx.

22. Reserve negative CHMx codes to CSS, customers. Change table to document that transitions are legal in all modes to IS=0 for interrupts and IS=1 for exceptions.

23. SCBB bit 30 is MBZ also.

24. Clarify the PCB stack pointers are not maintained.

25. Add invalid digit to reserved operand list.

26. Don't allow IPL>0 if current mode NEQ 0.

27. Document FC and FFFF as permanently reserved.

28. Document saved PC on all exceptions/interrupts.

29. Document that Access Violation takes precedence over Translation Not Valid.

30. Add interrupts example.

31. Machine check is on a best effort basis.

32. Move IPL here from chapter 9. IPL<31:5> is ignored on write, returned 0.

33. SIRR ignores <31:5>; ignores <4:0> = 0; reserved aborts if <4:0> GTRU 15.

34. Add table of arithmetic trap types to 6.3.

35. Change reserved operand reference from CVTNP to CVTTP, CVTSP (ECO).

36. Clarify FP and SP unpredictability on aborts.

37. Merge divide by zero trap; add index trap (INDEX ECO).

38. Note that faults do not restore everything, only enough.

39. Another inter vs except distinction is previous mode field.

40. CCodes may change on reserved operand.

41. STAR violates reset not clear halt reason.

42. If a routine sets FPD, UNPREDICTABLE.

43. SCBB is realy machine dependent.

44. Stack switch on except/int is optional if to same stack.

45. MTPR IPL changes states in state table.

46. Don't execute first instruction of interrupt routine if reinterrupt.

47. Check console halt with all interrupts.

48. Add that STAR memory errors on 1B.

49. Remove that interrupts can not push on stack. The future might find it useful.

50. Interrupts/exceptions can be to shared process space!

51. Add that STAR traps PC, @PC, -(PC).

52. MTPR, MFPR now get stack pointers.

53. Add bootstrap intial settings of these registers.

54. POLY underflow occurs at end.

55. Add processor register reserved operands.

56. CALL clears saved T.

57. Add documentation of xSP registers.

58. Legal to REI with FPD and TP both set. TP takes precedence.

59. SIRR<31:4> are ignored.

60. On STAR, vector<1:0> = 2 with no WCS is a HALT.

61. On STAR, CHMx vector<1:0> is ignored.

62. Check ASTLVL in REI only if returning to IS=0.

63. Add explanation and rationale for T and TP.

64. Add usage note that IPL splits with all ISP above all KSP.

65. Add usage note for ISRs to not drop IPL below original.

66. Add usage notes for debuggers.

Rev 2 to Rev 3:

1. Renumber interrupt levels, in hex, for hardware 10 to 1F and software 01 to 0F.

2. Provide names of Software Interrupt Request Register (SIRR) and Software Interrupt Summary Register (SISR).

3. Change "Numeric" to "Decimal".

4. Clarify fact that previous mode is cleared on interrupt, and loaded from current mode on exceptions.

5. Add description of software interrupt mechanism.

6. Add description of System Control Block (SCB), its base register (SCBB), and format of vectors.

7. Remove requirement to halt on exception if IS=1.

8. Combine cache parity error with machine check. Fix machine check to note that exception may be fault.

9. Change name of trace exception back to trap. Even though implemented as a fault, people think of it as a trap.

10. Fill out descriptions of arithmetic traps, giving type codes, result, and condition codes.

11. Eliminate decimal strings from reserved operand list, and illegal combinations in CALLx, RET; illegal PCB in LDPCTX; illegal register in MTPR, MFPR.

12. Combine memory error abort with machine check.

13. Remove concept of programmable device vectors.

Rev 1 to Rev 2:

1. Define IPL's -16 to 15, and clarify definitions.

2. Change all references to "EXCEPTION STACK" to "KERNEL STACK".

3. Add descriptions of processor modes and stacks.

4. Eliminate trap-pending bits other than TP.

5. Redefine PSL.

6. Delete references to ISL.

7. Rewrite exception/interrupt and REI flow descriptions to work with new scheme.

8. Simplify TBIT description for CHM scheme.

9. Eliminate address break and reserved address traps.

10. Rewrite REI description.

11. Add CHM instructions.

12. Add chart describing state transitions.

13. Fix introduction to eliminate misconception that exceptions are always synchronous and interrupts aren't. New version could still use word.

14. Remove concept of interrupt enables.

15. Clarify stack residency and validity.

16. Rename T bit to trace. Clarify how T and TP work for trace fault.

17. Define effect of bits 1:0 of vector to select kernel stack, interrupt stack, or WCS.

18. Redefine SCB vectors, eliminating and combining conditions, and add interrupt vectors.

19. Change most aborts to faults or traps.

20. Specify results from arithmetic traps; clarify floating overflow and underflow.

21. Change Master Control Block (MCB) to System Control Block (SCB).

22. Include checks of compatability mode conditions in PSL of REI.

23. Add example showing serialization of trap, interrupt, and trace.

[End of SR6R6.RNO]

CHAPTER 6

EXCEPTIONS AND INTERRUPTS

31-Jan-79 -- Rev 6

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, 01 to 0F), and 16 hardware levels (10 to 1F, hex). User applications, system calls, and system services all run at process level, which may be thought of as IPL 0. 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 01 through 0F (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 6.2.3). Once a software interrupt request is made, it will be cleared by the hardware when the interrupt is taken.

Interrupt levels 10 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 20:16 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.

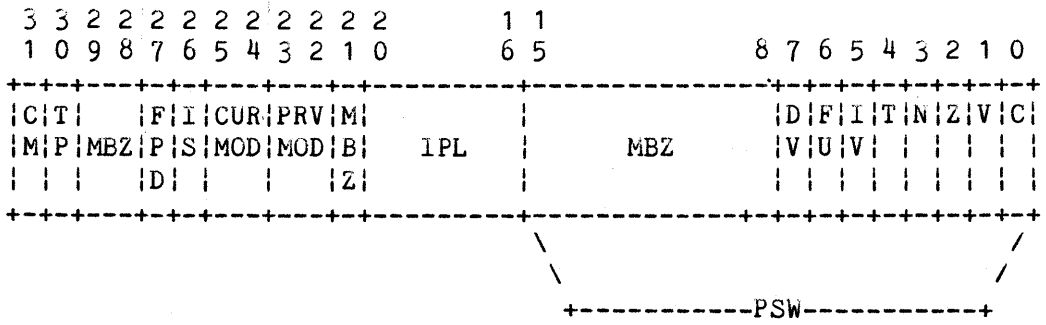
7. The previous mode field in the PSL is always set to Kernel on an interrupt, but on an exception it indicates the mode of the exception.

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.



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 between instructions (before examining T), a trace fault is taken. The effect is that setting bit 4 forces a trace ~~trap~~ fault before the execution of each subsequent instruction. When 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.)
15:9	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 the instruction addressed by PC cannot simply be restarted, and must be resumed 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 interrupted instruction's execution are UNPREDICTABLE. If a routine sets FPD, the results are also UNPREDICTABLE.

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.

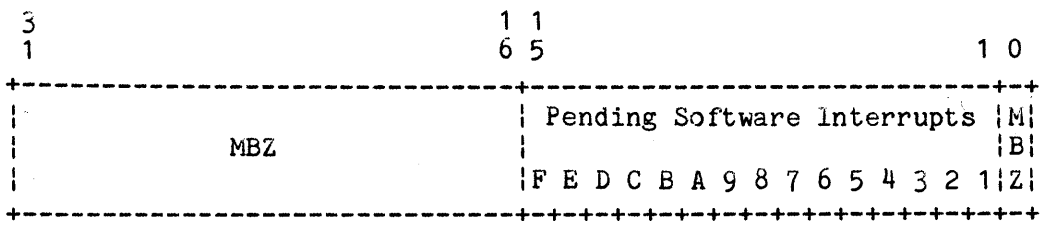
Interrupt level 1E (hex) is reserved for power fail. Interrupt level 1F (hex) is reserved for those exceptions that must lock out all processing until handled. This includes the hardware and software "disasters" (machine check and kernel stack not valid). It might also be used to allow a kernel mode 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.



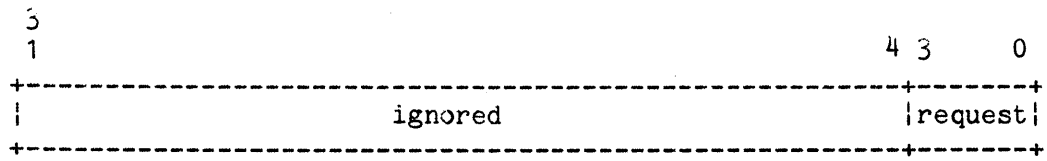
Software Interrupt Summary Register

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 ~~five~~ four bit privileged register used for making software interrupt requests.



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.

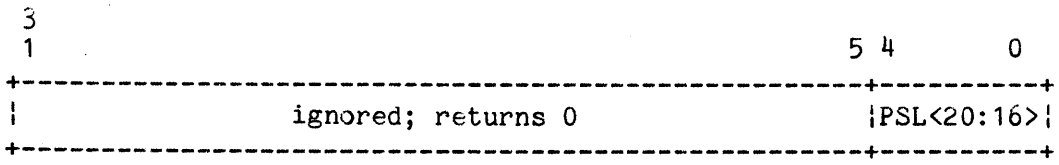
2. The requester uses MTPR src,#SIRR to request an interrupt at the appropriate level.

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.



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 (see 6.13). 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	contents of IPL (hex)	state after event SISR (hex)	IPL in PSL on stack
(initial)	5	0	0
MTPR #8,#IPL	8	0	0
MTPR #3,#SIRR	8	8	0
MTPR #7,#SIRR	8	88	0
MTPR #9,#SIRR interrupts to	9	88	8,0
device interrupts to	11	88	9,8,0
device service routine REI	9	88	8,0
IPL9 service routine REI	8	88	0
MTPR #5,#IPL changes IPL to 5 and the request for 7 is granted immediately	7	8	5,0
IPL7 service routine REI	5	8	0
initial IPL5 service routine REI back to IPL0 and the request for 3 is granted immediately	3	0	0
IPL3 service routine REI	0	0	--

6.4 EXCEPTIONS

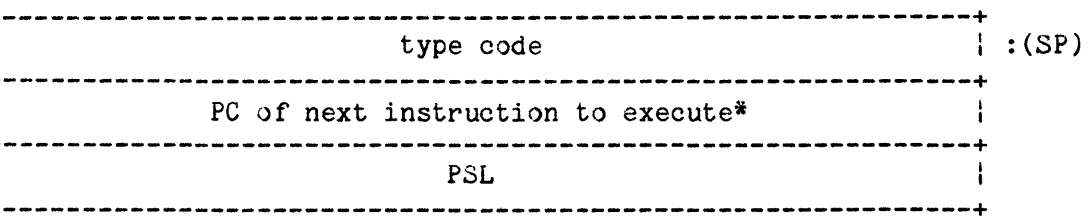
Exceptions can be grouped into six classes:

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

A trace fault for ... [illegible] ... other exceptions. If multiple exceptions can occur for an instruction, the order in which they are taken is UNPREDICTABLE

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. An appropriate distinguishing code is pushed on the stack as a longword:



*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

\Floating overflow, underflow, divide by zero traps occur only on the original VAX-11/780. On all other VAX processors, these floating arithmetic exception conditions result in faults.\

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 and condition codes are UNPREDICTABLE. The zero divisor can be either +0 or -0.

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. 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 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. The destination operand is unaffected. The saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. The type code pushed on the stack is A (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 (ABORT; see chapter 4 for restartability)
3. POLY degree too large (FAULT)
4. Decimal string too long (FAULT)
5. Invalid digit in CVTTP, CVTSP (FAULT)

6. Bit field too wide (FAULT)

7. Invalid combination of bits in PSL restored by REI (FAULT)

8. Reserved pattern operator in EDITPC (ABORT; 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
POBR<1:0> NEQU 0
POBR LSSU 2**31
POBR 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
POLR<31:27>'POLR<23:22> NEQU 0
P1LR<30:22> NEQU 0
ASTLVL<2:0> GTRU 4
```

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:

0	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 10, 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 6.7). 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 (except that a service routine invoked by CHMx and terminated by REI is considered a single instruction). The saved PC on a trace is the address of the next instruction that would normally be executed.

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) (see section 6.3). 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, the PSL<T> bit is defined to produce a trap after any other traps or aborts. 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. ~~See section 6.11 for detailed flows~~.

The rules of operation for trace are:

1. At the beginning of an instruction, if T is set then TP is set.

2. If the instruction faults or an interrupt is serviced, the pushed PSL<TP> is cleared. The pushed PC is set to the start of the faulting or interrupted instruction.

3. If the instruction aborts or takes an arithmetic trap, the pushed PSL<TP> is set or cleared as the result of step 1.

4. 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 the pushed PSL<TP> is set or cleared as the result of step 1.

5. At the beginning of an instruction, if TP is set then a trace pending fault is taken.

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. REI will trap either if T was set when the REI was executed or if TP in the saved PSL is set. Because of this, the instruction sequence CHMx...REI acts as a single instruction. Note that the trace exception occurring after an REI that had TP set before being executed will be taken with the new PSL. Thus, special care must be taken if exception or interrupt routines are traced.

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 (see 6.6.4).

The detection of interrupts and other exceptions occurs ~~before~~ the ~~detection of a trace exception. However, this causes~~ no difficulties ~~since~~ 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)	Trace exception 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	(trap after 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	(only one trap)
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 reqained 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 CMHx 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 <1:0> for Kernel Stack Not Valid is 3, the behavior of the processor is UNDEFINED (see section 6.5.2).

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.

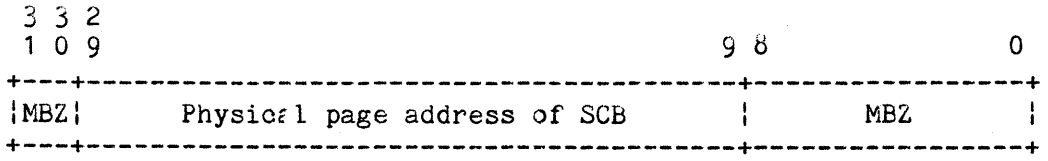
6.4.6.3 Machine Check Exception - A machine check exception indicates that the processor detected an internal error in itself. As usual for exceptions, this exception is taken independent of IPL. IPL is raised to 1F (hex) only if vector<1:0> is 1.. Implementation specific information is pushed on the stack as longwords. The processor specifies the number of bytes pushed by placing the number of bytes pushed as the last longword pushed. (0 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 <1:0> for machine check is 3, the behavior of the processor is UNDEFINED (see section 6.5.2).

6.5 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.

6.5.1 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.



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.

6.5.2 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. See also 6.10.

1. Service this event on the interrupt stack. If this event is an exception, the IPL is raised to 1F (hex). See also 6.10.

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, see section 6.13. 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
00	Unused			Reserved to DIGITAL.
04	Machine Check	Abort/ Fault/ Trap	*	Processor-and error-specific information is pushed on the stack, if possible. Restartability is processor specific. If vector<1:0> 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.
08	Kernel Stack Not Valid	Abort	0	Serviced on the interrupt stack (i.e. IS <- 1). IPL is raised to 1F (hex).
0C	Power Fail	Interrupt	0	IPL is raised to 1E (hex).
10	Reserved/Privileged Instruction	Fault	0	Opcodes reserved to DIGITAL and privileged instructions.
14	Customer Reserved Instruction	Fault	0	XFC instruction.
18	Reserved Operand	Fault/ Abort	0	Type depends on circumstances. See ~~6.5.4.~~ 6.4.3.2
1C	Reserved Addressing Mode	Fault	0	
20	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	0	
2C	Breakpoint Instruction	Fault	0	
30	Compatibility	Fault/ Abort	1	A type code is pushed onto the stack. See 6.~~6.3.~~ 4
34	Arithmetic	Trap/ Fault	1	A type code is pushed onto the stack. See 6.4.
38-3C	Unused			Reserved to DIGITAL.
40	CHMK	Trap	1	The operand word is sign extended and pushed onto the stack. Vector<1:0> MBZ.
44	CHME	Trap	1	The operand word is sign extended and pushed onto the stack. Vector<1:0> MBZ.
48	CHMS	Trap	1	The operand word is sign extended and pushed onto the stack. Vector<1:0> MBZ.
4C	CHMU	Trap	1	The operand word is sign extended and pushed onto the stack. Vector<1:0> MBZ.
50-80	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-F4	Unused			Reserved to DIGITAL
F8	Console Terminal Rec.	Interrupt	0	IPL is 14 (hex).

FC Console Terminal Trans. Interrupt 0 IPL is 14 (hex).

100-1FC Device Vectors Interrupt 0

In the VAX-11/780 processor, only hardware 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

6.6 STACKS

At any time, the processor is either in a process context (IS=0) 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<1:0> = 1) and interrupts below this boundary run on the kernel stack (vector<1:0> = 0). Typically, AST delivery (IPL 2) is on the kernel stack and all higher levels are on the interrupt stack.

6.6.1 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.

6.6.2 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 M
;;TTY12: - PLEASE DELETE ANY UNNECESSARY FILES ON DSKB....THANX...
OVZWL), 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.

6.6.3 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	0	ISP
0	0	KSP
0	1	ESP
0	2	SSP
0	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 <1:0> of the vector for the event as follows:

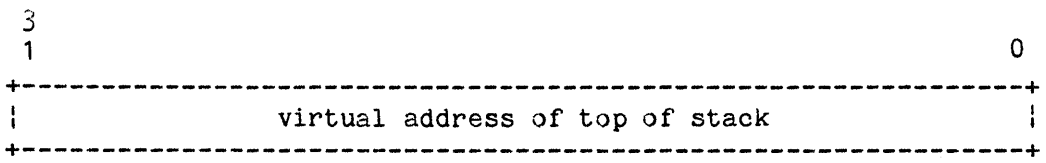
		vector<1:0> 00	01
PSL<IS>	0	KSP	ISP
	1	ISP	ISP

Values 10 (binary) and 11 (binary) of the vector<1:0> are used for other purposes. Refer to section 6.5.2 for details.

6.6.4 Accessing Stack Registers

The processor implements five privileged registers to allow access to each stack pointer. These registers always access the specified pointer even for the current mode. 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 in the PCB, MTPR and MFPR of these registers access the PCB. The register numbers were chosen to be the same as PSL<26:24>. The previous stack pointer is the same as PSL<23:22> unless PSL<IS> is set. At bootstrap time, the contents of all stack pointers are UNPREDICTABLE.



Kernel Stack Pointer	KSP = 0
Executive Stack Pointer	ESP = 1
Supervisor Stack Pointer	SSP = 2
User Stack Pointer	USP = 3
Interrupt Stack Pointer	ISP = 4

6.7 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, it gets an arithmetic trap, and an interrupt request is recognized, the following sequence occurs:

1. The instruction finishes, storing all its results.

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 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 trap service routine runs, and exits with REI, which finds a PSL having TP=1.

7. The trace occurs, again pushing PC and PSL but this time with TP=0.

8. Trace service routine runs, and exits with REI.

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

        if {arith trap needed and no other abort
                or trap} then {initiate arith trap};

    1$: {possibly take interrupts or console halt};
        if PSL<TP> EQLU 1 then      !if trace pending, take trace fault
                begin
                PSL<TP> <- 0;
                {initiate trace fault};
                end;

        {possibly take interrupts or console halt};
        PSL<TP> <- PSL<T>;          !if trace enable, set trace pending

        {go start instruction execution};
        !Note: 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};
                !note: all instructions end by flowing
                ! through 1$, thus the REI from the service
                ! routine will return to 1$
                end;
```

6.8 INITIATE EXCEPTION OR INTERRUPT

<none> Initiate Exception or Interrupt

Operation:

```
!The notation PSL<xxx>_SP is used to refer
! to the SP appropriate to the mode xxx specified
! in the PSL. The actual stack pointer may be either
! in the PCB or in a general register (see chapter 7).

{disable interrupts};
tmp1 <- SCB[vector];            !get correct vector
if tmp1<1:0> EQLU 3 then {UNDEFINED};
if tmp1<1:0> EQLU 0 AND {machine check or
                kernel stack not valid} then {UNDEFINED};
if tmp1<1:0> NEQU 0 AND {CHMx} then {UNDEFINED};
if tmp1<1:0> EQLU 2 then
        begin
        if {writable control store exists and is loaded}
                then {enter writable control store}
                else {UNDEFINED};
        end;
if PSL<IS> EQLU 0 then                  !switch stacks
        begin
        PSL<CUR_MOD>_SP <- SP; !save old SP
        if tmp1<1:0> EQLU 1 then
                SP <- ISP;
                else
                SP <- new_mode_SP; !kernel_SP unless CHMx
        end;
tmp2 <- PSL;
PSL<CM,TP,FPD,DV,FU,IV,T,N,Z,V,C> <- 0; !clean out PSL
if {interrupt} then
        PSL<PRV_MOD> <- 0;              !kernel mode
        else
        PSL<PRV_MOD> <- PSL<CUR_MOD>;
PSL<CUR_MOD> <- new_mode;       !kernel_mode unless CHMx
-(SP) <- tmp2;          !on a fault or abort, the saved
                        !condition codes are UNPREDICTABLE
-(SP) <- PC;            !as backed up, if necessary
{push parameters if any};
                        !if kernel stack not valid exception
                        !occurs while pushing tmp2, PC, or other
                        !parameters then PSL <- tmp2 before
                        !initiating exception
if {interrupt} then                     !set new IPL
        PSL<IPL> <- new_IPL
        else
        if tmp1<1:0> EQLU 1 then
                PSL<IPL> <- 31;         !1F (hex)
if tmp1<1:0> EQLU 1 then PSL<IS> <- 1;  !otherwise keep old IS
PC <- tmp1<31:2> ' 0<1:0>;              !longword aligned
```

```
        {enable interrupts};
        if {PSL<IPL> LEQU 15} AND {PSL<IPL> GEQU 1} then
                SISR<PSL<IPL>> <- 0;
```

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, a trace pending fault, 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
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 0} OR
   {tmp2<IS> EQLU 1 AND tmp2<CUR_MOD> NEQU 0} OR
   {tmp2<IS> EQLU 1 AND tmp2<IPL> EQLU 0} OR
   {tmp2<IPL> GTRU 0 AND tmp2<CUR_MOD> NEQU 0} OR
   {tmp2<PRV_MOD> LSSU tmp2<CUR_MOD>} OR
   {tmp2<IPL> GTRU PSL<IPL>} OR
   {tmp2<PSL_MBZ> NEQU 0} then {reserved operand fault};
if {tmp2<CM> EQLU 1} AND
   {{tmp2<FPD,IS,DV,FU,IV> NEQU 0} 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 0 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};
```

Condition Codes:

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

Exceptions:

```
reserved operand
```

Opcodes:

```
02    REI     Return from Exception or Interrupt
```

Description:

A longword is popped from the current stack and held in a temporary 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 6.10.3). 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.

4. ~~See Chapter 10~~
When entering compatibility mode, general
regists

CHM Change Mode

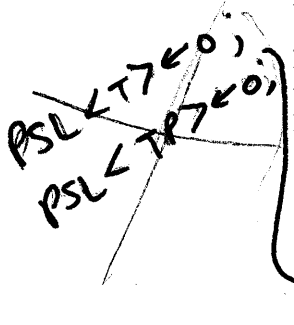
Purpose: request services of more privileged software

Format:

```
opcode  code.rw
```

Operation:

```
tmp1 <- {mode selected by opcode (K=0, 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 40+tmp1*4 (hex) as SCB offset
        using tmp4 as the new SP
        and not storing SP again};
```



Condition Codes:

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

```
halt
```

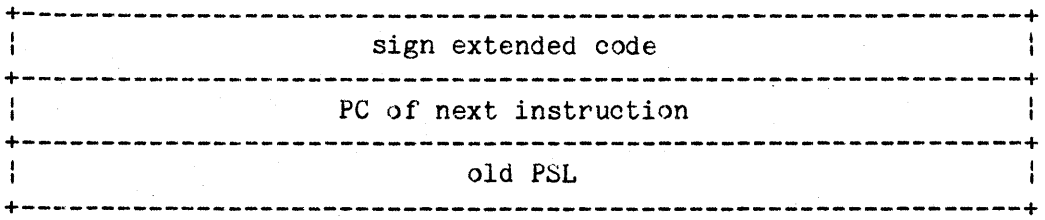
Opcodes:

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

Description:

Change Mode instructions allow processes to change their access mode in a controlled manner. The instruction 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:



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<1:0> code NEQU 0 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=0 IPL=0	Super IS=0 IPL=0	Exec IS=0 IPL=0	Kernel IS=0 IPL=0	Kernel IS=0 IPL>0	Kernel IS=1 IPL>0	Program Halt
User IS=0 IPL=0	CHMU REI	CHMS	CHME	CHMK Excep(0)	Inter(0)	Excep(1) Inter(1)	impos- sible
Super IS=0 IPL=0	REI*	CHMU,S REI	CHME	CHMK Excep(0)	Inter(0)	Excep(1) Inter(1)	impos- sible
Exec IS=0 IPL=0	REI*	REI*	CHMU,S,E REI	CHMK Excep(0)	Inter(0)	Excep(1) Inter(1)	impos- sible
Kernel IS=0 IPL=0	REI*	REI*	REI*	CHMUSEK REI* Excep(0) MTPR IPL LDPCTX	MTPR IPL Inter(0)	SVPCTX Excep(1) Inter(1)	HALT Instr.
Kernel IS=0 IPL>0	REI*	REI*	REI*	MTPR IPL REI*	CHMUSEK REI* Excep(0) Inter(0) MTPR IPL LDPCTX	SVPCTX Excep(1) Inter(1)	HALT Instr.
Kernel IS=1 IPL>0	REI*	REI*	REI*	REI*	LDPCTX REI*	SVPCTX REI Excep Inter MTPR IPL	HALT Instr. CHMUSEK

Inter is Interrupt (0) is vector<1:0> = 0; see 6.9
Excep is Exception (1) is vector<1:0> = 1; see 6.9

* Any REI that increases mode can cause an interrupt request at IPL 2 for AST delivery.

Processor State Transitions

[End of Chapter 6]

Title: VAX-11 Process Structure -- Rev 4

Specification Status:

Architectural Status: under ECO control

File: SR7R4.V07

PDM #: not used

Date: 10-Aug-78

Superseded Specs:

Author: D. Hustvedt

Typist: N. Ford, B. Call

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 7 describes the set of hardware primitives provided in VAX-11 that permit the implementation of high performance process dispatching software. It defines the process context known to hardware, the process control block (PCB) and the handling of software interrupts termed asynchronous system traps (ASTs).

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original issue	Hastings	Oct-75
Rev 2	Revised for software needs	Hustvedt	Jan-76
Rev 3	Total Rewrite	Hustvedt	8-Jun-76
Rev 4	Fixes to LDPCTX and SVPCTX	Conklin/Taylor	10-Aug-78

Rev 3 to Rev 4:

1. Typos.
2. Add privileged register descriptions of ASTLVL and PME, including their initial values.
3. ASTLVL GEQU 5 is reserved.
4. Contents of PCBB is initially UNPREDICTABLE.
5. Change "privileged instruction" to "reserved instruction."
6. Change R12, R13 to AP, FP.
7. Scheduler runs on IPL = 3.
8. Add software offsets for PCB.
9. Add notes to LDPCTX and SVPCTX.
10. Add reserved operand checks to LDPCTX.

Revision 3 is a total rewrite replacing all previous revisions.

1. All process structure instructions described in previous revisions are deleted as well as the descriptions of firmware implemented scheduling. These changes were made as the result of efforts to simplify the VAX architecture and reduce implementation risks. The primitives described in this revision are believed to be sufficient to implement the same functionaity in software.
2. Deleted CHMI instruction
3. Changed LDPCTX to effect transition to Kernel Stack.
4. Changed SVPCTX to include function of CHMI.
5. Add translation buffer purge to LDPCTX.
6. Changed interrupt priority level numbering to eliminate negative IPL.

[End of SR7R4.RNO]

CHAPTER 7

PROCESS STRUCTURE

10-Aug-78 -- Rev 4

7.1 PROCESS DEFINITION

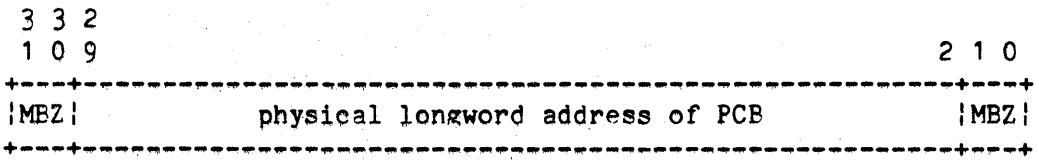
A process is a single thread of execution. It is the basic schedulable entity that is executed by the processor. A process consists of an address space and both hardware and software context. The hardware context of a process is defined by a Process Control Block (PCB) that contains images of the 14 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 P0BR, P0LR, 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 set of 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.

PCBB



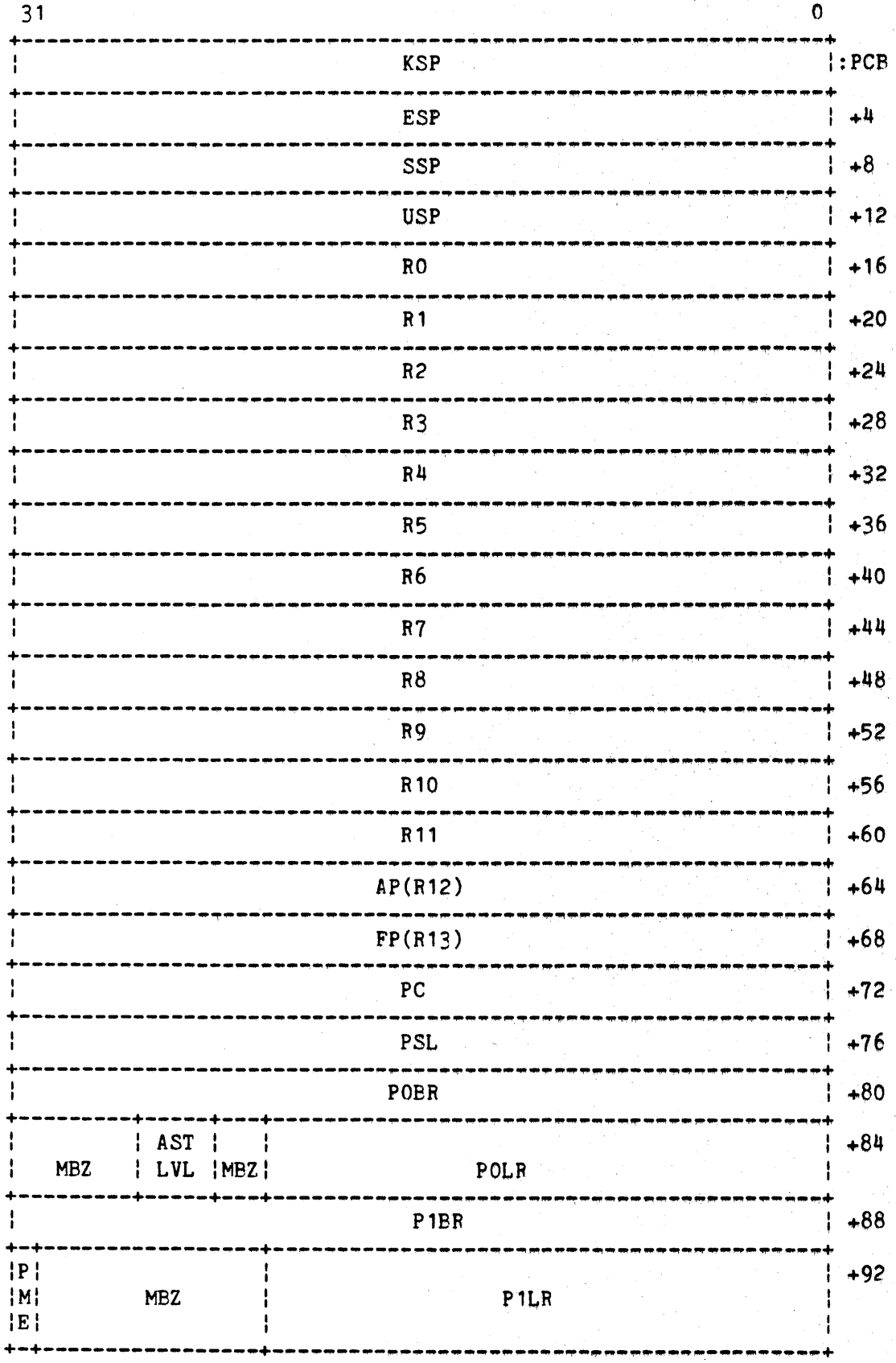
(read/write)
Process Control Block Base

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.

Process Control Block (PCB)



Description of Process Control Block

Longword	Bits	Mnemonic	Description
0	<31:0>	KSP	Kernel Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 0 and IS = 0.
1	<31:0>	ESP	Executive Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 1.
2	<31:0>	SSP	Supervisor Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 2.
3	<31:0>	USP	User Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 3.
4-17	<31:0>	R0-R11, AP,FP	General registers R0 through R11, AP, FP.
18	<31:0>	PC	Program Counter.
19	<31:0>	PSL	Program Status Longword.
20	<31:0>	P0BR	Base register for page table describing process virtual addresses from 0 to 2**30-1. See chapter 5.
21	<21:0>	P0LR	Length register for page table located by P0BR. 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
0	AST pending for access mode 0 (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	<31:0>	P1BR	Base register for page table describing process virtual addresses from 2**30 to 2**31-1. See chapter 5.
23	<21:0>	P1LR	Length register for page table located by P1BR. Describes effective length of page table. See chapter 5.
23	<30:22>	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 permit their behavior to be observed without interference from other system activity.

Software symbols are defined for these locations by using the prefix PTX$L_ and the mnemonic shown above. For example, the PCB offset to R3 is PTX$L_R3. The following are also defined:

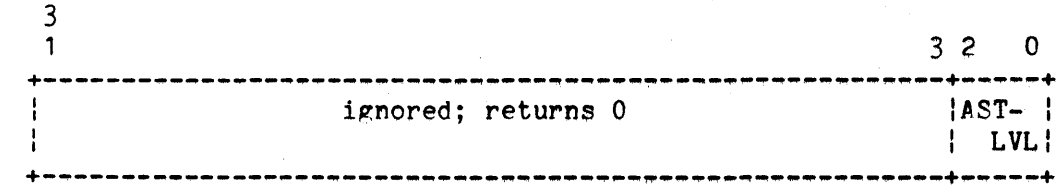
PTX$L_POLRASTL longword 21

PTX$L_P1LRPME longword 23

To alter its P0BR, P1BR, P0LR, 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.

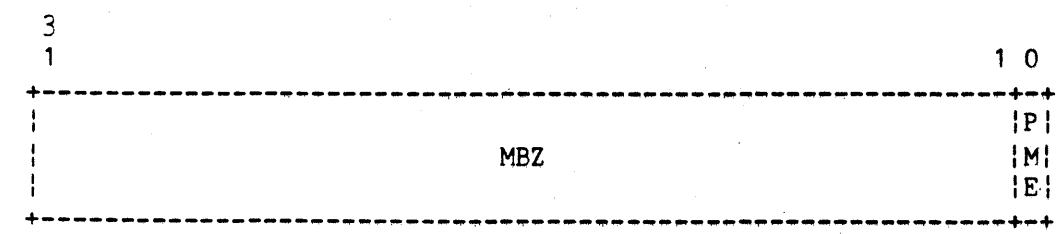
7.2.3 Process Privileged Registers

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.



(read/write)
AST Level Register

An MTPR src,#ASTLVL with src<2:0> GEQU 5 results in a reserved operand fault. At bootstrap time, the contents of ASTLVL is 4.



(read/write)
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 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). Each 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 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 0
        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;
R0 <- (PCB+16);
R1 <- (PCB+20);
R2 <- (PCB+24);
R3 <- (PCB+28);
R4 <- (PCB+32);
R5 <- (PCB+36);
R6 <- (PCB+40);
R7 <- (PCB+44);
R8 <- (PCB+48);
R9 <- (PCB+52);
R10 <- (PCB+56);
R11 <- (PCB+60);
AP <- (PCB+64);
FP <- (PCB+68);
tmp1 <- (PCB+80);
if {tmp1<31:30> NEQU 2} OR {tmp1<1:0> NEQU 0} then
                {reserved operand abort};
P0BR <- tmp1;
if (PCB+84)<31:27> NEQU 0 then {reserved operand abort};
if (PCB+84)<23:22> NEQU 0 then {reserved operand abort};
P0LR <- (PCB+84)<21:0>;
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<31:30> NEQU 2} OR {tmp2<1:0> NEQU 0} then
                {reserved operand abort};
P1BR <- tmp1;
if (PCB+92)<30:22> NEQU 0 then {reserved operand abort};
P1LR <- (PCB+92)<21:0>;
PME <- (PCB+92)<31>;
if (PCB+92)<30:22> NEQU 0 then {reserved operand abort};
if PSL<IS> EQLU 1 then
```

P0BR P1BR may be checked as pure chapter

OK

```
                    begin
                    ISP <- SP;
                    {interrupts off};
                    PSL<IS> <- 0;
                    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:

06	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 that do, LDPCTX loads the internal registers from the PCB. Those 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
        {interrupts off};
        PSL<IS> <- 1;
        SP <- ISP;
        {interrupts on};
        end;
```

Condition Codes:

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

Exceptions:

privileged instruction

Opcodes:

07	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 that do, SVPCTX stores the internal registers into the PCB. Those 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 is intended to illustrate how the process structure instructions can be used to implement process dispatching software. It is assumed that this simple dispatcher 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
```

[End of Chapter 7]

Title: VAX-11 System Architectural Implications -- Rev 4

Specification Status:

Architectural Status: Under ECO Control

File: SR8R4.V09

PDM #: not used

Date: 10-Aug-78

Superseded Specs:

Author: D. Rodgers

Typist: J. Bess

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 8 discusses system structural implications and implementation constraints of the VAX-11 architecture. The broad categories of interaction are: data sharing and synchronization, restartability, interrupts and errors. Chapter 8 also discusses I/O in a general way.

Revision History:

Rev #	Description	Author	Revised Date
Rev 3	Original Version	D. Rodgers	2-Jun-76
Rev 4	Document cache and interlocks	Conklin/Taylor	30-Jul-78

Rev 3 to Rev 4:

1. Move I/O here from chapter 9.
2. Document interlocks on ADAWI.
3. Document cache and its constraints.
4. xxxQUE are aligned (ECO).
5. Ensure that fault when interlocked does not hang.
6. Note multiproc rule for update system PTEs.
7. Clean up discussion of interlocks.

Rev 3:

Original creation

[End of SR8R4.RNO]

CHAPTER 8

SYSTEM ARCHITECTURAL IMPLICATIONS

10-Aug-78 -- Rev 4

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.

8.1 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 0 and 1 contain the values 5 and 6. Suppose one processor executes INCB 0 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. Three instructions (BBSSI, BBCCI, ADAWI) 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-11/780, the SBI primitive operations are interlock read and interlock write.

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 aligned longword reads and writes in an uninterruptible sequence to allow queues to be maintained without other interlocks in a uniprocessor 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, floating, ~~double 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 F).

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.

8.2 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).

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.

8.3 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. In order that software may dependably access peripheral device registers, instructions used to access them must not permit device interrupts during their execution.

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).

8.4 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.

8.5 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.

8.6 I/O STRUCTURE

8.6.1 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 ~~Volume~~ 1). 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.

8.6.2 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, double, floating and field references in the I/O space result in UNDEFINED behavior.

[End of Chapter 8]

Title: VAX-11 Privileged Register and Console Structure -- Rev 4

Specification Status:

Architectural Status: Under ECO control

File: SR9R4.V11

PDM #: not used

Date: 4-Aug-78

Superseded Specs: CH9R3.SRM

Author: C. Learoyd, P. Conklin

Typist: J. Sleet

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 9 is divided into two major areas; the privileged register space; and various architecturally defined console functions. To ensure consistency across members of the VAX architecture, certain console functions are defined architecturally. The second section of Chapter 9 describes the operation of HALT, CONTINUE, INITIALIZE, and some restrictions on other console features. Also, a minimum console functionality is defined, and a description of the system bootstrap mechanism is given.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	Rodgers	10-Oct-75
Rev 2	Added Registers	Rodgers	21-Feb-76
Rev 3	Added Console Specs	Learoyd	3-Jun-76
Rev 4	Add Processor Reg Specs	Conklin/Taylor	4-Aug-78

Rev 3 to Rev 4:

1. Remove I/O section to chapter 8.
2. MT/FPR of xSP always works (Stack Register ECO).
3. Move IPL register to chapter 6.
4. Add SID definition.
5. Add Console terminal definition.
6. Add Clock definition.
7. Move TBIA/TBIS to chapter 5.
8. Move VAX-11/780 error registers to chapter 11.
9. Document VAX-11/780 accelerator registers.
10. Document VAX-11/780 Control store registers.
11. MTPR faults if RO; MFPR faults if WO.
12. Add register numbers.
13. Add VAX-11/780 registers to table.
14. IPL is R/W not W.
15. ICR is RO, NICR is WO; not R/W.
16. Add ASTLVL in 7; MAPEN in 5; PMR in 7; TODR in 9.
17. Add Initial values of all registers, T Buf, cache.
18. Add Boot set of general registers.
19. Make console terminal mandatory.
20. SID<23:0> now type specific.
21. Add RO, RW, WC, WO field notations.
22. Clarify identity of function for machines having the same register number.
23. Console halt only between instruction; not an interrupt.
24. Change reset to initialize.
25. change i buf clear from deposit to continue.

26. Console virtual relative to PSL<CUR_MOD>.

27. Add system restart section.

28. Move xSP to chapter 6.

29. Move process registers to chapter 7.

30. Clarify when RXCS and TXCS cause interrupts.

31. Make ACCS type be 8 bits.

32. Note diagnostics use NOP for scope sync.

33. Deposit/virtual command maintains PTE.

34. Add 780 SID spec.

35. Name change CRxx, CTxx to RXxx, TXxx.

36. Missing TODR is always 0.

Rev 2 to Rev 3:

1. Added notion of UNIBUS space.

2. Added I/O register constraint rules.

3. Added I/O space interlocking.

4. Added MTPR, MFPR descriptions.

5. MTPR, MFPR require kernel mode.

6. Added clocks, bootstrap (no spec yet).

7. Added WCS reg (no spec yet)

8. Added FCS Internal Registers.

9. Added console function section.

10. Deleted description of multiple register sets.

11. Make WCS registers implementation-specific.

12. Delete PSL in PIR space.

13. Add discussion of per-process registers in privileged register space.

14. Added description of interval timer.

15. Add description of time-of-day clock

Rev 1 to Rev 2:

1. Added list of privileged registers
2. Expanded I/O description
3. Expanded privileged register description.

[End of SR9R4.RNO]

CHAPTER 9

PRIVILEGED REGISTERS AND CONSOLE

4-Aug-78 -- Rev 4

9.1 INTRODUCTION

Chapter 9 is divided into two major areas; the privileged register space; and various architecturally defined console functions. To ensure consistency across members of the VAX architecture, certain console functions are defined architecturally. The second section of Chapter 9 describes the operation of HALT, CONTINUE, INITIALIZE, and some restrictions on other console features. Also, a minimum console functionality is defined, and a description of the system bootstrap mechanism is given.

9.2 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. See section 9.2.3 for a description of these instructions.

All the internal processor registers are summarized in the tables at the end of this section. Those which need futher explanation are described below. Reference to general registers means R0 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.

9.2.1 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.

Furthermore, it is possible that implementations which retain some or all registers only in the PCB will implement the MTPR and MFPR for those registers as a no-op, at least in the sense that the destination or register is not written. To ensure that the PCB is always correctly updated and to permit implementation flexibility, software must use the following convention when referencing any of the per-process registers in the processor register space.

1. MTPR - Software must first write the value directly into the proper location in the current PCB by using a MOVL (for example) then execute an MTPR with the same source as the MOVL. Implementations which do not retain internal copies of these registers may effectively no-op the MTPR instruction. They must not, however, take a reserved operand fault which would normally occur for a non-existent register.

2. MFPR - Software must first read the value directly from the proper location in the current PCB by using a MOVL (for example), then execute an MFPR instruction using the same destination as the MOVL. Implementations which do not retain internal copies of these registers may effectively no-op the MFPR instruction. They must not, however, take a reserved operand fault which would normally occur for a non-existent register.

9.2.2 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). However, software may access any of the four stack pointers not currently selected by the current mode and IS bits in the PSL via the MTPR and

MFPR instructions. Results are UNPREDICTABLE 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 applies only to the KSP and ISP, since these instructions require kernel privilege).

9.2.3 The MTPR And MFPR Instructions

MTPR Move To Processor Register

Format:

```
opcode  src.rl, procreg.rl
```

Operation:

```
if PSL <CUR_MOD> NEQ 0 then {reserved
        instruction fault};
PRS[procreg] <- src;
```

Condition Codes:

```
N <- src LSS 0;    !if register is replaced
Z <- src EQL 0;
V <- 0;
C <- C;


N <- N;            !if register is not replaced (see 9.2.1)
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 may occur or the instruction may no-op (see 9.2.1).

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

MFPR Move From Processor Register

Format:

```
opcode procreg.rl, dst.wl
```

Operation:

```
if PSL <CUR_MOD> NEQ 0 then {reserved
        instruction fault};
dst <- PRS[procreg];
```

Condition Codes:

```
N <- dst LSS 0;    !if destination is replaced
Z <- dst EQL 0;
V <- 0;
C <- C;


N <- N;            !if destination is not replaced (see 9.2.1)
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 may occur or the instruction may no-op (see 9.2.1).

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

VAX-11 Series Registers

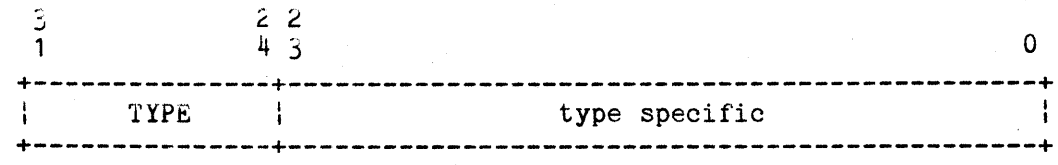
Register Name	Mnemonic	Number	Type	Scope	Init?
Kernel Stack Pointer	KSP	0	R/W	PROC	--
Executive Stack Pointer	ESP	1	R/W	PROC	--
Supervisor Stack Pointer	SSP	2	R/W	PROC	--
User Stack Pointer	USP	3	R/W	PROC	--
Interrupt Stack Pointer	ISP	4	R/W	CPU	--
P0 Base Register	P0BR	8	R/W	PROC	--
P0 Length Register	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

VAX-11/780 Specific Registers

Register Name	Mnemonic	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

9.3 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.



(read only)

System Identification Register

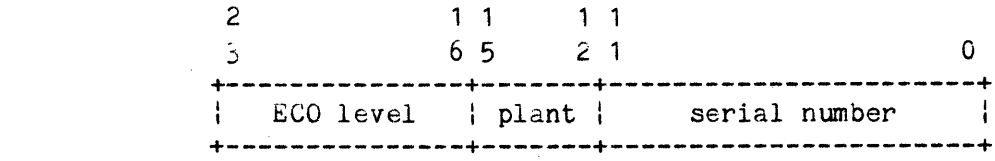
Type A unique number assigned by engineering to identify a specific processor:

2 = COMET
3 = NEBULA
4 = 11/780 with floating fault

0 = Reserved to DIGITAL (error)
1 = VAX-11/780
2 through 127 = Reserved to DIGITAL
128 through 255 = Reserved to CSS and customers

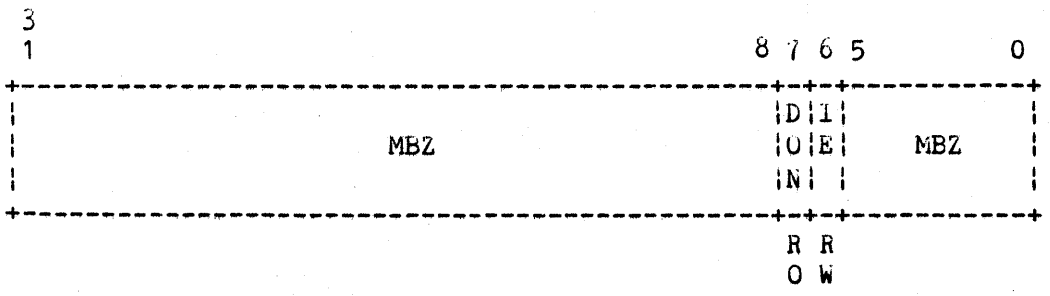
type specific format and content is a function of the value in type. It is intended to include such information as serial number and revision level.

For the VAX-11/780, the type specific format is:

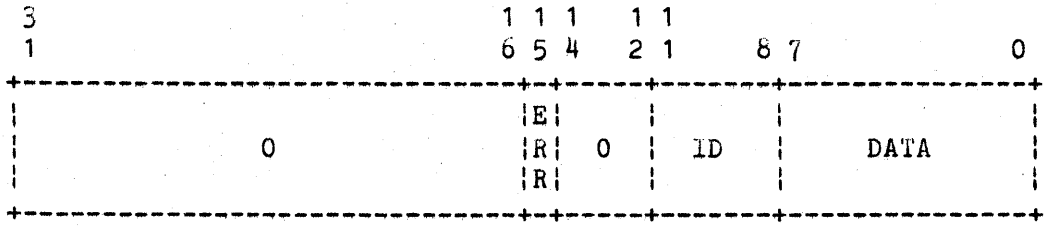


9.4 CONSOLE TERMINAL REGISTERS

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.

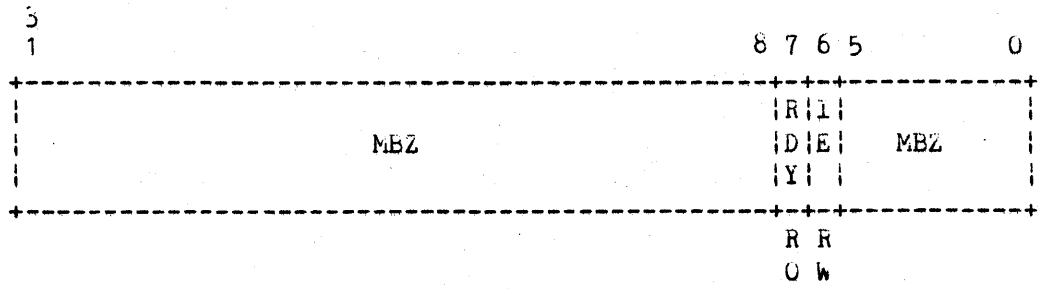


Console Receive Control/Status (RXCS)

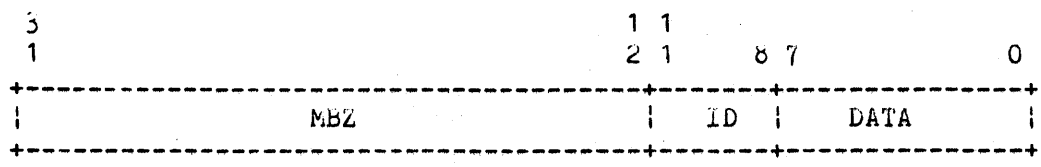


(read only)
Console Receive Data Buffer (RXDB)

At bootstrap time, RXCS is initialized to 0. 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 20. 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 0 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 0 then the data is from the console terminal. If ID is non-zero then the entire register is implementation dependent.



Console Transmit Control/Status (TXCS)



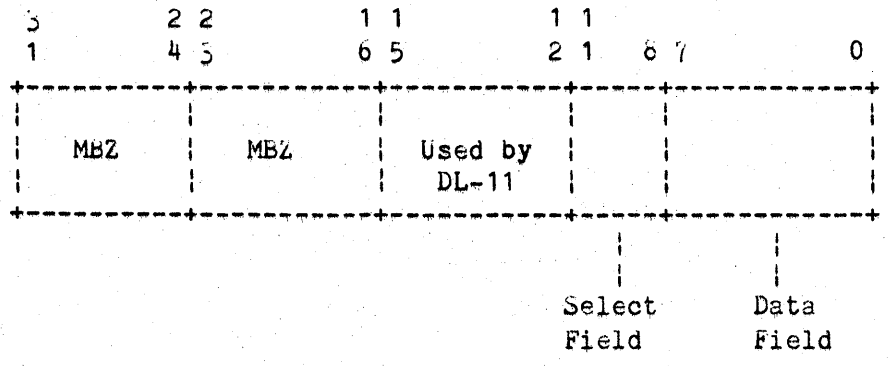
(write only)
Console Transmit Data Buffer (TXDB)

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 20. 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 0 to 1). The software can send a datum by writing it to DATA. When a MTPR src,#TXDB is expected, RDY is cleared as is any interrupt request. If ID is written 0 then the datum is sent to the console terminal. If ID is non-zero then the entire register is implementation dependent.

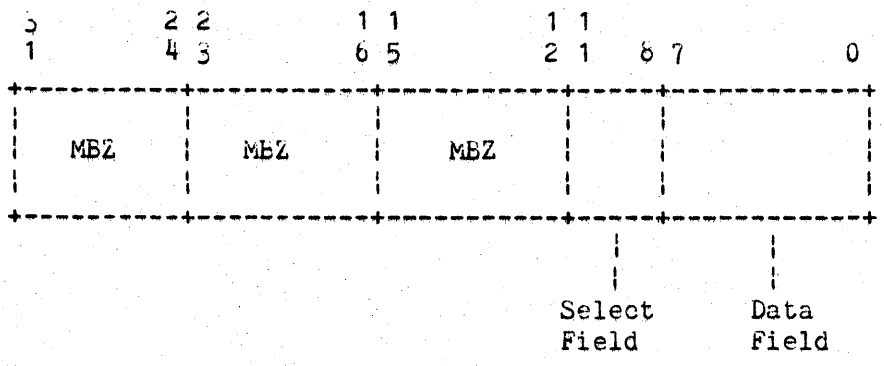
On the VAX-11/780 if ID is one then the datum is sent to the floppy disk.

9.4.1 VAX-11/780 Console Register Implementation

RXDB



TXDB



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

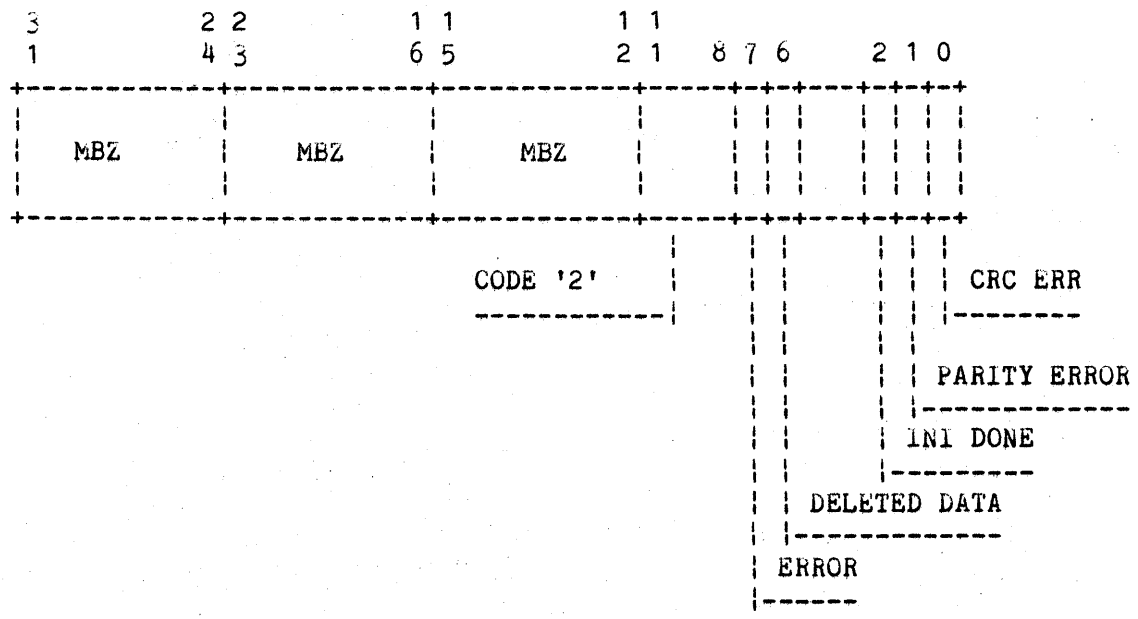
Code 5 (Protocol Error), is sent by the console when one of the following occurs:

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.

9.4.1.1 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



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 CLOCK REGISTERS

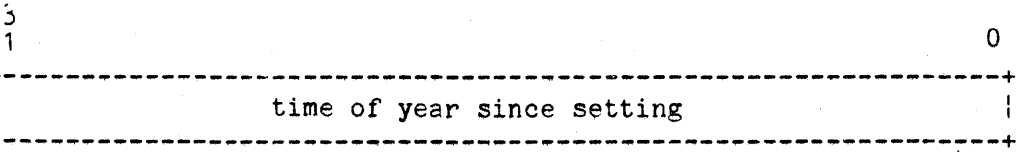
The clocks consist of an optional time of year clock and a mandatory interval clock. The time of year clock is used to measure the duration of power failures and is only required for unattended restart after a power failure. The interval clock is used for accounting, for time dependent events, and to maintain the software date and time.

9.5.1 Time-of-Year Clock (optional)

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 .0025% accuracy (approximately 65 seconds per month). The counter has a 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. The least significant bit of the counter represents a resolution of 10 milliseconds. Thus, the counter cycles to 0 after approximately 497 days.

If the battery has failed, so that time is not accurate, then the register is cleared upon power up. It then 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.

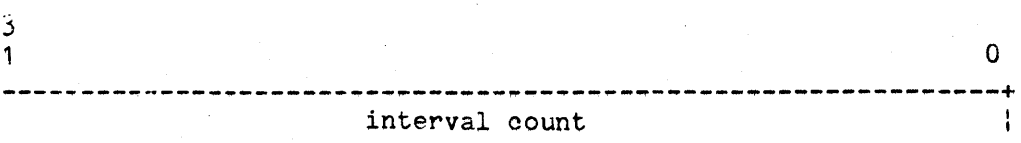


Time of Year (TODR)

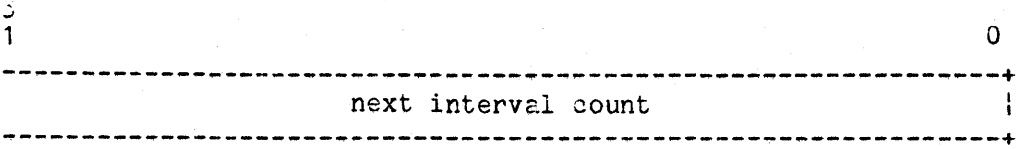
If the clock is not installed, then the clock always reads out as 0 and ignores writes.

9.5.2 Interval Clock

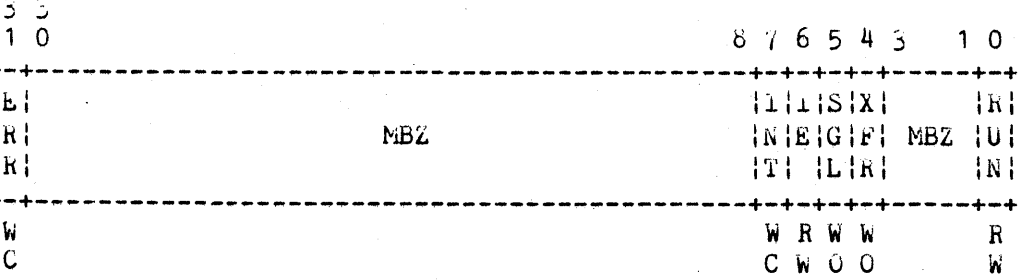
The interval clock provides an interrupt at IPL 24 at programmed intervals. The counter is incremented at 1 microsecond intervals, with at least .01% accuracy (8.64 seconds per day). The clock interface consists of three registers in the privileged register space:



interval count register (ICR)



next interval (NICR)



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 0 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.

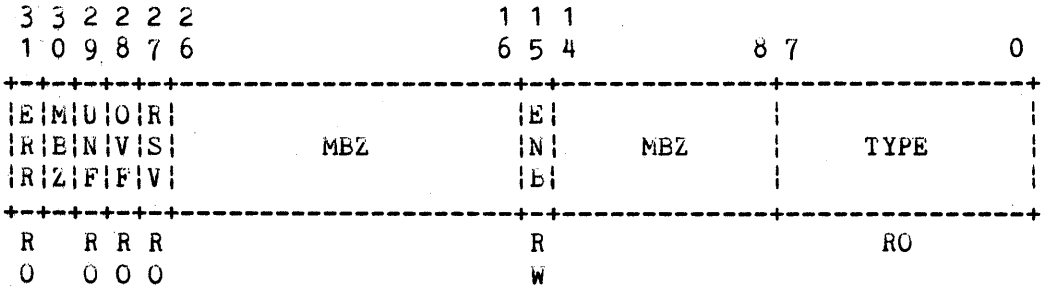
Thus, to setup the interval clock, load the negative of the desired interval into NICR. Then a MTPR #^X51,#ICCS will enable interrupts, reload ICR with the NICR interval and set run. Every "interval count" microseconds will cause INT to be set and an interrupt to be requested. The interrupt routine should 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 <0> of ICCS are cleared. The rest of ICCS and the contents of NICR and ICR are UNPREDICTABLE.

9.6 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.



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.

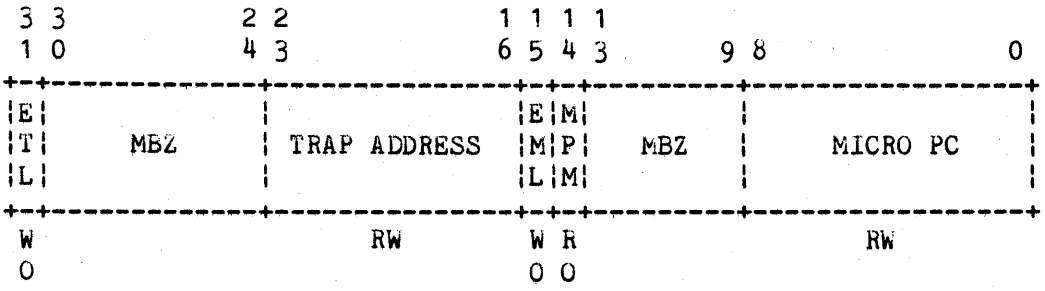
OVF <28> Read only bit specifying that the last operation had an overflow.

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.



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.

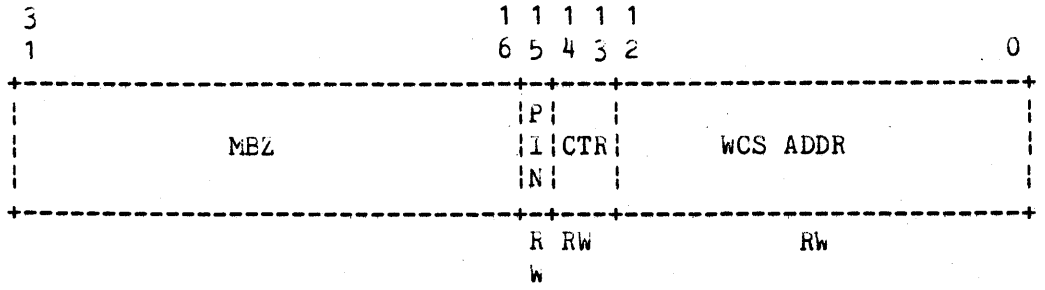
EML <15> ENABLE MICRO PC MATCH LOAD. A write only bit that when set causes <8:0> to be loaded into the accelerator's micro PC match register.

TRAP <16:23> TRAP ADDRESS. A read/write field used by the main processor to force the accelerator to a specified micro location.

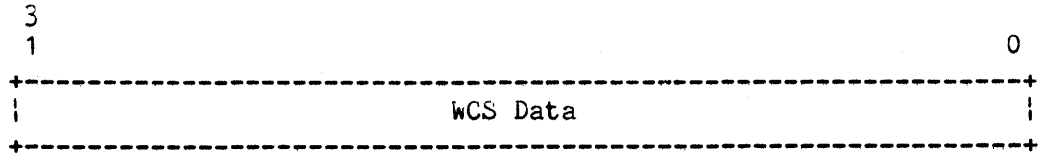
ETL <31> ENABLE TRAP ADDRESS LOAD. A write only bit that when set causes <23:16> to be loaded into the accelerator's trap address register. Subsequently, the main processor's micro code can force the accelerator to trap to this location by asserting an internal signal.

9.7 VAX-11/780 MICRO CONTROL STORE

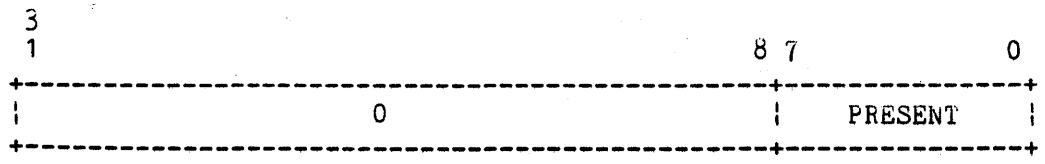
The VAX-11/780 processor has three registers for control/status of its microcode. Two are used for writing into any writable control store (WCS) and one is used to control micro breakpoints.



Writable Control Store Address (WCSA)

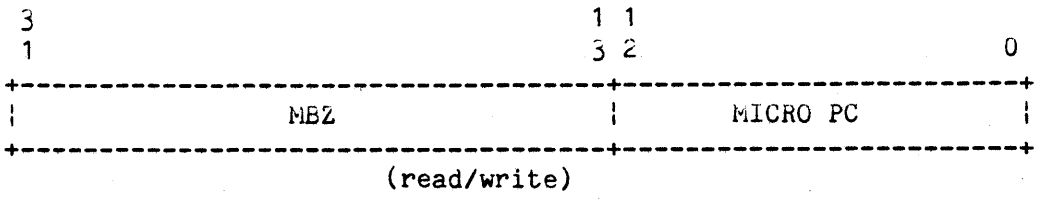


(on Write)



(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.



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 CONSOLE FUNCTIONS

CPU consoles by nature are very implementation specific. However, the specification and interaction of certain console functions are defined architecturally for consistency across VAX-11 series members. Also, a minimum console functionality is defined.

Console functions may be split into three broad areas: system operator interaction including communication with the operating system; control functions, such as START and HALT; and maintenance functions such as examine, deposit. The control functions are generally used for both system operation and maintenance. Although it is recognized that sophisticated programmers may use the maintenance features for software debugging, this is not a constraint on the console implementations. Consoles may have write access to machine registers and other hardware facilities not accessiblethrough the instruction set even in Kernel mode. It may be possible for the console to leave the machine in an inconsistent state. In this case machine behavior is UNDEFINED.

9.6.1 Operator Interaction

A set of four registers is reserved in the privileged register space for a DL-11-like interface to a terminal. If the system implements a console with a terminal, this interface will be implemented and communicate with the console terminal. Other uses of the terminal, for example as a system operator's console, are optional.

9.8.2 Control Functions

9.8.2.1 Halts - There are essentially three types of CPU halts: error halts, console requested halts (i.e., a console HALT function) and a halt as the result of a HALT instruction being executed. A CPU halt is defined as stopping any further instruction execution and entering a state that will recognize and service any console requested functions, or execute an auto-boot sequence.

1. Error halts - An error halt may occur at any stage of instruction execution as a result of certain error conditions, for example interrupt stack not valid. The CPU will set the appropriate bit in the CPU error register and enter the console service state. On a best efforts basis, error halts do not alter the value of the PSL, the ISP, or the PC. Following an error halt the PC is left pointing to either the opcode of the instruction that caused the error, or, if between instruction execution (e.g., responding to an interrupt), to the next executable instruction.

2. Console requested halts - The CPU halts at the beginning of the next instruction following the halt request, but does not execute any of that instruction. The PC is pointing to that instruction. It is also possible that an exception or interrupt occured at the same time as the halt request. If this is the case then the PC will point to the first instruction of the service routine. Note that instructions that execute indefinitely are interruptible during execution. Thus, the regular clock interrupts ensure that a console halt request will be honored except possibly for problems handling urgent interrupts.

3. HALT instruction execution - The HALT instruction executes by invoking a halt request similar to that caused by the console. The PC is left pointing to the next instruction.

9.8.2.2 Continue - If the CPU is running, a continue function has no effect. If the CPU is halted at the beginning of an instruction, continue resumes execution with the instruction currently pointed to by the PC. The instruction buffer, if implemented, will be flusned before execution is resumed.

A continue following an error halt without an intervening initialize results in UNDEFINED operation. On the VAX-11/780, this is another halt.

9.8.2.3 Initialize - A console initialize function initializes the processor to the state defined in section 9.9, bootstrapping, except that all general registers (R0 through R13, SP, and the PC) are not intialized.

9.8.2.4 Start - A start sequence consists of an initialize function followed by loading the PC, followed by a continue. Implementations that do not provide a mechanism to load the PC will set the PC to a constant value as a result of the initialize function. The specific machine documentation includes this value.

9.8.3 Maintenance Functions

Maintenance functions include features such as examine and deposit, single micro cycle, etc. Many of those functions are highly implementation dependent. However, certain areas require architectural specification.

9.8.3.1 Examine And Deposit -

1. Functions that do not explicitly alter the programmer visible state (e.g., examine or deposit) will not change any of the programmer visible state other than, in the case of deposit, the location explicitly referenced and its map entry.

2. Error conditions, such as non-existent memory, that would normally cause an abort or fault, will be inhibited during console service memory cycles (i.e., examine and deposit), and no error status bits set in the CPU error register. The CPU will, however, have a mechanism to inform the console that the error has occurred. Any status from an error halt may be lost.

3. Console virtual memory accesses have the privilege of the PSL's current access mode and will occur in the current process's virtual address space. A virtual memory deposit will cause the modify bit in the page table entry to be set.

9.8.3.2 Single Instruction - A "single instruction" function is not defined architecturally. However, clearing the console halt request along with setting a flag which will cause the console request to be asserted at the next "beginning of instruction" effectively yields a "single instruction" functionality.

9.8.4 Minimum Console

All VAX-11 series members have a mechanism to implement at least the following console functions:

1. Halt
2. Continue
3. Initialize

In addition a means for indicating that the CPU is running (i.e., not halted) and a "power on" indication will be provided.

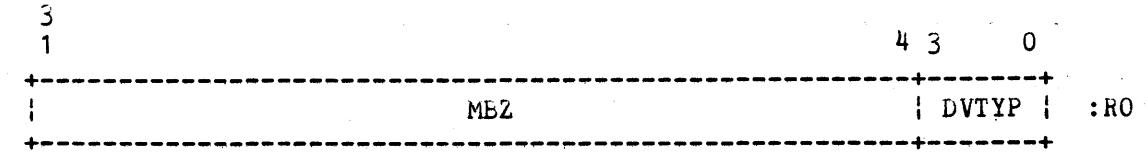
9.9 SYSTEM BOOTSTRAPPING

At bootstrap time the machine is initialized as follows:

R0 through R13	boot parameters
PC	boot code
PSL	041F0000 (hex)
translation buffer	UNPREDICTABLE
cache	empty or valid
instruction buffer	empty or valid
KSP,ESP,SSP,USP,ISP	UNPREDICTABLE
P0BR,P0LR,P1BR,P1LR	UNPREDICTABLE
SBR,SLR	UNPREDICTABLE
PCBB,SCBB	UNPREDICTABLE
IPL	1F (hex)
ASTLV	4
SISR	0
ICCS	UNPREDICTABLE except <6>'<0> clear
NICR,ICR	UNPREDICTABLE
TODR	time since set or since power up
RXCS	0
RXDB	UNPREDICTABLE
TXCS	80 (hex)
MAPEN	0
PMR	0
SID	System identification
ACCS	0 if no accelerator; 8001 (hex) if floating point accelerator
ACCR	UNPREDICTABLE
WCSA	UNPREDICTABLE
WCSD	writable blocks
SBIxx	\TBS\
MBRK	UNPREDICTABLE

As part of the primary (ROM) bootstrap, certain general registers are set up to specify the bootstrap device. These registers pass the

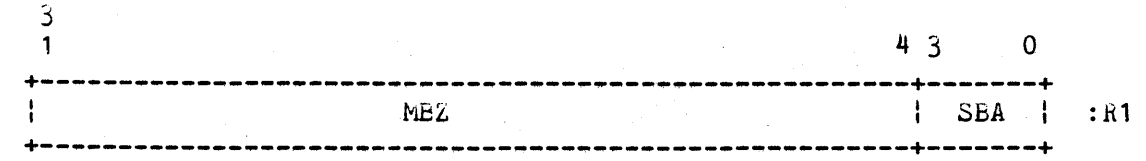
bootstrap information to any further (software) bootstraps. If a register specified is not applicable, 0 is specified. The boot register parameters are:



DVTYP identifies the device type as follows:

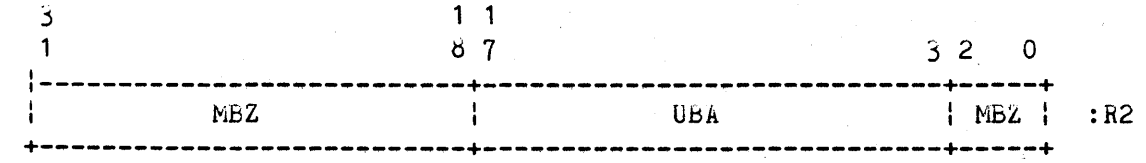
0 - disk pack
1 - cartridge disk
4 - magnetic tape
5 - DECtape
8 - paper tape
12 - Synchronous Communications Line
14 - Asynchronous Communications Line

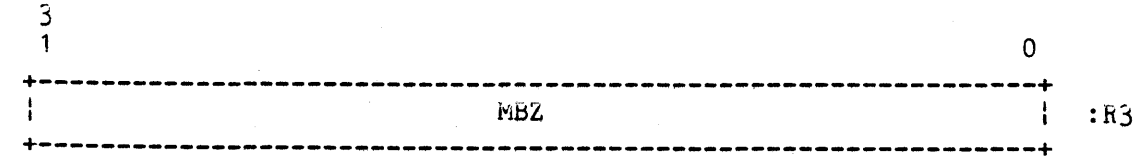
All other values are reserved to DIGITAL. \ **** Need CSS device reservation. **** \



SBA is an implementation dependent system bus address. On VAX-11/780 SBA is the NEXUS on the SBI.

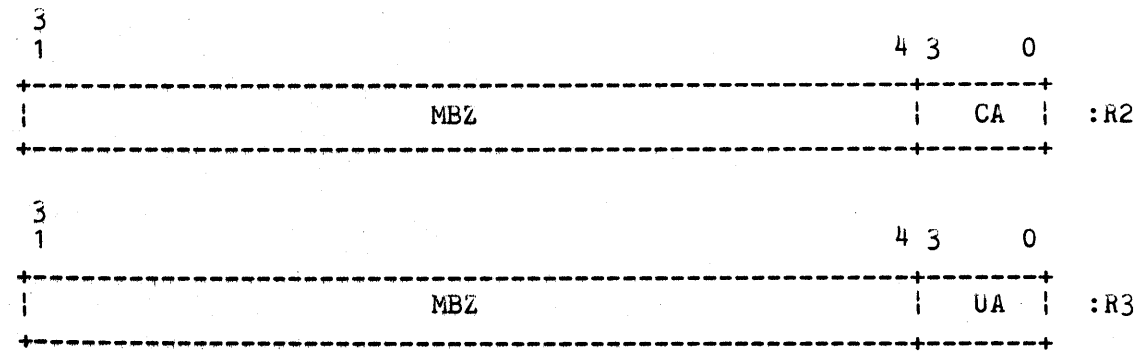
For a UNIBUS bootstrap device,





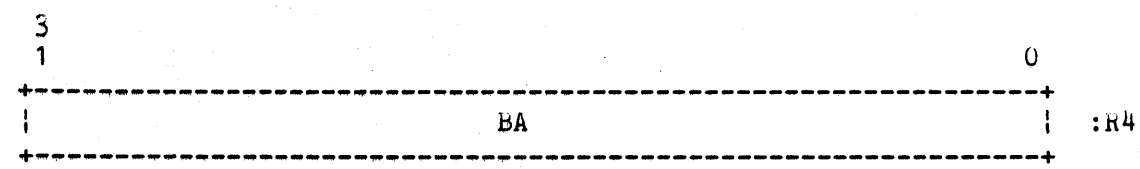
UBA is UNIBUS address bits <17:3>.

For a MASSBUS bootstrap device,



CA is the formatter number. UA is the unit number.

For all devices,



BA is the block address on the device.

The contents of registers R5 through R13 are UNPREDICTABLE.

9.10 SYSTEM RESTART

After restoration of power or optionally after the processor halts, system operation is resumed by initializing the processor, locating the lowest addressed memory, and starting at the address specified in the longword at this lowest address plus 4. Software depends upon the content of the three longwords at lowest address, lowest address plus 4 and lowest address plus 8, and also that the memory configuration does not change from system interruption to restart.

[End of Chapter 9]

Title: VAX-11 PDP-11 Compatibility Mode -- Rev 4

Specification Status: Fully approved

Architectural Status: under ECO control

File: SR10R4.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs:

Author: D. Cutler

Typist: J. Bess

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Chapter 10 describes the Compatibility Mode that is provided in the VAX architecture to allow a certain subset of PDP-11 programs to be directly executed on VAX machines. VAX compatibility mode hardware, in conjunction with a compatibility mode software executive, can emulate the environment provided to user programs on a PDP-11.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Distributed	Rothman	25-Sep-75
Rev 2	ECO 3	Rothman	1-Jan-76
Rev 3	April Meeting	Cutler	3-Jun-76
Rev 4	Typos	Cutler	28-Feb-77

Rev 3 to Rev 4:

1. Typos

Rev 2 to Rev 3:

The chapter has been extensively modified due to the restartibility of instructions and the new memory management architecture. Virtually all sections were affected. The following are major revisions:

1. The PDP-11 environment exclusion list was expanded to include more features that are not supported by compatibility mode hardware.

2. MF/TP(I/D) were moved to the list of supported instructions. They execute exactly like push and pop instructions and ignore previous mode.

3. Compatibility mode can now be entered only via an REI instruction since the process structure has been deleted from the architecture. The number of privileged bits in PSL has been reduced due to the restartibility of instructions.

4. Compatibility mode programs directly m
;;TTY12: - PLEASE TAKE NOTE OF NOTICE.TXT.
ap into the first 64k
bytes of the per process part of the address space without any special address manipulations.

5. Exceptions now push information on the kernel stack and ISL is no longer present in the architecture.

6. The new memory management architecture obviated the need for the MCMA instruction.

Rev 1 to Rev 2:

1. The chapter has been extensively rewritten. Included now are complete definitions of the interfaces to VAX native mode, including memory management and exceptions. A set of notes has also temporarily been added at the end of the chapter that answers some common questions, describes open issues, and explains rejected alternatives to some of the compatibility mode specifications. The other items in this list are only the changes to revision 1 of this chapter, not the additions.

2. RTI and RTT no longer perform the same operations in compatibility mode. They are now exactly equivalent to their PDP-11 counterparts.

3. The comment about floating point simulation has been changed since the VAX mode floating point instructions have changed.

4. The Move Compatibility Mode Address instruction has been changed to make it completely general with respect to restriction level.

[End of SR10R4.RNO]

CHAPTER 10

PDP-11 COMPATIBILITY MODE

28-Feb-77 -- Rev 4

Compatibility Mode is provided in the VAX architecture to allow a certain subset of PDP-11 programs to be directly executed on VAX machines. 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 from a complete PDP-11 the normal operation of the following features:

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. (Compatibility mode programs can directly reference I/O devices if and only if proper mapping has been established by VAX mode software.)
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).
10. Floating point instructions.

This chapter is split in two parts. The first part is a brief description of the PDP-11 environment provided by the VAX compatibility mode hardware. Details of the operation of PDP-11 compatible operations can be found in the appropriate PDP-11 handbook. The second part describes the hardware mechanisms provided in the VAX architecture which enable the implementation of various compatibility

mode executives.

10.1 COMPATIBILITY MODE USER ENVIRONMENT

10.1.1 General Registers And Address Modes

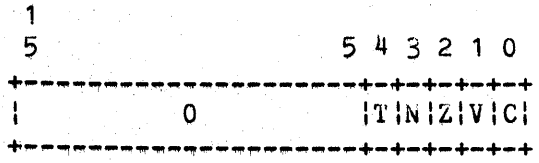
All of the PDP-11 general registers and addressing modes are provided in compatibility mode. Side effects caused by a destination address calculation have no effect on source values, and auto-increment modes in JMP and JSR do not affect the new PC. All addresses are 16 bits wide.

10.1.2 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. See Sections 10.5 and 10.7.

10.1.3 Processor Status Word

A subset of the full PDP-11 Processor Status Word is available in compatibility mode. The format of the compatibility mode PSW is:



The PSW can only be affected by the condition code instructions, RTI, and RTT. When an RTI or RTT instruction is executed, bits 15 through 5 in the saved PSW on the stack are ignored.

10.1.4 Instructions

The following instructions are provided by the compatibility mode hardware.

TABLE 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)
.057DD	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
074RSS	XOR
077RNN	SOB
.1SSDD	MOV(B)
.2SSDD	CMP(B)
.3SSDD	BIT(B)
.4SSDD	BIC(B)
.5SSDD	BIS(B)
06SSDD	ADD
16SSDD	SUB

* 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.

The following trap instructions cause the machine to enter VAX mode, where either the complete trap may be serviced, or where just the instruction may be simulated. See Section 10.5.

TABLE 2
Compatibility Mode Trap Instructions

Opcode (octal)	Mnemonic
000003	BPT
000004	IOT
104000-104377	EMT
104400-104777	TRAP

The following instructions and all other opcodes not defined above are considered reserved instructions in compatibility mode, and trap to VAX mode. See Section 10.5.

TABLE 3
Compatibility Mode Reserved Instructions

Opcode (octal)	Mnemonic
000000	HALT
000001	WAIT
000005	RESET
00023N	SPL
0064NN	MARK
07500R	FADD--FIS
07501R	FSUB--FIS
07502R	FMUL--FIS
07503R	FDIV--FIS
17XXXX	FP11 Floating Point

Note that no floating point instructions are included in compatibility mode.

10.2 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 10.6 for a complete description of how T bit traps work in compatibility mode.

VAX 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 to compatibility mode, those bits must be zero, or a reserved operand fault occurs.

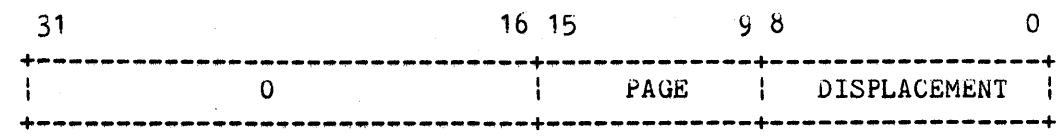
10.2.1 General Register Usage

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 and the stacked R15 (PC) are zero. Since there are no FP11 floating point instructions in compatibility mode, there are no floating accumulators.

10.3 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:



The PDP-11 capability of providing different access protection to different segments is automatically provided since protection is specified at the page level in the VAX architecture (i.e., VAX pages are smaller than PDP-11 segments).

The memory management system protects and relocates compatibility mode addresses in the normal 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 which can be caused by memory management in VAX mode can also occur when relocating a compatibility mode address. See Chapter 5.

Most of the features of the KT11-D affecting the user environment can be simulated with the VAX memory management system. The following table provides a general description of how this can be done; reference Chapter 5 of this manual and the appropriate PDP-11 documents for details of each system.

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 space 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.

What follows is an example of how a PDP-11 environment can be created using the above concepts. 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

10.4 COMPATIBILITY MODE EXCEPTIONS AND INTERRUPTS

All interrupts and exception conditions which 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 following exception conditions are specific to compatibility mode. All these exceptions create a three longword frame on the kernel stack containing PSL, PC, and one longword of trap specific information. Bits 15 through 0 of this longword contain a code indicating the specific type of trap and bits 31 through 16 are zero.

10.4.1 Reserved Instruction Trap

These are the opcodes that are defined as reserved in compatibility mode. The code for the reserved instruction trap is 0.

10.4.2 BPT Instruction

The code for the BPT instruction is 1.

10.4.3 IOT Instruction

The code for the IOT instruction is 2.

10.4.4 EMT Instruction

The code for the group of EMT instructions is 3.

10.4.5 TRAP Instruction

The code for the group of TRAP instructions is 4.

10.4.6 Illegal Instructions

Illegal instructions in compatibility mode are JMP and JSR instructions with a register destination. The code for illegal instructions is 5.

10.4.7 Odd Address Error

An odd address error trap is caused in compatibility mode whenever a word reference is attempted on a byte boundary. References that use the SP or PC are always word references, even if used in a byte instruction. The code for odd address errors is 6.

10.5 T BIT OPERATION IN COMPATIBILITY MODE

A compatibility mode T bit trap occurs at the end of an instruction when the T bit is set in the PSL at the beginning of the instruction. A T bit trap also occurs at the end of an RTI instruction if the T bit was set in the PSW POPed from the stack. On T bit traps, a two long word kernel stack frame is created, containing PSL and PC. IPL and IS are zero and CM is one in the stacked PSL. Compatibility mode T bit trap uses the same vector as VAX mode T bit trap. See Chapter 6.

\There have been problems with the operation of the T bit on PDP-11s. The T pending bit in the PSL solves those problems. For that reason, the operation of the T bit in compatibility mode is not identical to that of PDP-11s.\

There are two ways to get the T bit set at the beginning of a compatibility mode instruction:

1. An RTT instruction is executed in compatibility mode and 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 T bit trap is taken after that instruction.

2. An REI instruction is executed in VAX mode which has both the T bit and CM bit set (and T pending clear) in the saved PSL image on the stack. Again, one instruction is executed, and the T bit trap is taken. (For a complete description of the interaction of REI, T bit, and T pending, 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 at the beginning of any compatibility mode instruction which does not trap.

   In this case, a T bit trap is taken after the instruction. The saved PSL has the T bit set and TP clear. The compatibility mode executive will do one of the following things:

   1. If it services the trap directly, it may clear the T bit in the saved PSL on the kernel stack if it no longer wants to trace the program, or it may leave it set if it wants to continue tracing the program. It exits with an REI.

   2. If it returns the trap to compatibility mode, it pushes a (16 bit) PC and (16 bit) PSW with the T bit set on the User stack to simulate the effect of the T bit trap. It then clears the T bit in the saved PSL image on the kernel stack, and does an REI. The compatibility mode service routine then may clear the T bit in the PSW image on its stack, as a function of whether or not it wants to continue tracing. The compatibility mode routine returns with RTT. (If it always clears the T bit in the saved PSW, it does not matter if it returns with RTI or RTT.)

2. T bit set at the beginning of an RTI or RTT.

   A T bit trap occurs immediately after the 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 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. This is the same case as for nontrapping instructions.

3. T bit set at the beginning of any instruction which causes a compatibility mode trap.

   The trap condition is serviced first. TP is set in the saved PSL pushed on the kernel stack. These traps may be serviced in one of two ways:

1. The compatibility mode executive directly services the trap condition.

   In this case, when the compatibility mode executive is done, it executes an REI. The TP and CM bits in the PSL image on the stack will be set, so a compatibility mode T bit trap will immediately be taken.

2. The compatibility mode executive returns the trap condition to a compatibility mode routine which services the trap.

   In this case, the compatibility mode executive will push a (16 bit) PC and (16 bit) PSW on the user stack to simulate the effect of the trap. The PSW pushed by the compatibility mode executive will have the T bit set, because the TP bit was set in the saved PSL on the kernel stack. The compatibility mode executive will then clear the T and TP bits in the saved PSL and do an REI to the compatibility mode service routine. When the compatibility mode routine is done servicing the trap, it will do an RTI, which will then cause the compatibility mode T bit trap to occur.

10.6 UNIMPLEMENTED PDP-11 TRAPS

There are several traps that occur in PDP-11s that 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 trap conditions such as power failure, memory parity, and memory management traps cause the machine to enter VAX mode.

10.7 COMPATIBILITY MODE I/O REFERENCES

Since I/O devices are accessible with all instructions in VAX mode (as in the PDP-11), I/O devices may be referenced directly from compatibility mode, if the memory mapping is set up to allow it. This may be done by mapping pages directly to I/O devices. Note that, in general, I/O devices will NOT appear in the physical address space on VAX machines the same way they do on PDP-11s, so existing PDP-11 programs that directly reference I/O devices probably will not work. In addition, compatibility mode programs can only do word or byte references; many VAX I/O devices may require that some references be 32 bits wide.

10.8 PROCESSOR REGISTERS

The only processor register available in compatibility mode is part of the PSW, and it may only be referenced with the condition code instructions, RTI, and RTT. Access to all other registers must be done in VAX mode.

10.9 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.

10.10 NOTES

\

1. There are no references to specific PDP-11 implementations. At some point, the PDP-11 "Table of Programming Differences" should be updated to include VAX compatibility mode. No one is committed to do this.

2. There have been proposals to make the upper four bits of the compatibility mode PSW be ones, so that compatibility mode programs appear to be running with the current mode and previous mode as User. This does not make any sense; the hardware never materializes the compatibility mode PSW, it only uses an image of it for the RTI and RTT instructions. That image is created by software; if it is desired to make the upper four bits one, the software may do it.

3. It must be made clear that all compatibility mode processes also have a VAX mode. There is nothing that prevents the VAX mode of the process from being rather extensive; the only constraint is that the VAX part of the process be careful about using virtual addresses 0 to FFFF(hex) since those are used for mapping the compatibility mode addresses.

4. A proposal to ignore the PSL bits that cause an UNDEFINED operation was rejected. That would potentially add more cost to the hardware, and it should not affect the software, since the hardware always returns those bits as zero in the PSL pushed on the exception stack when leaving compatibility mode.

5. A proposal to simulate KT11 registers MMR0 and MMR1 in hardware has been rejected. MMR0 and MMR1 can easily be simulated by software with the information available on memory management faults and aborts (i.e., MMR0 is the address of the instruction causing the fault and MMR1 is always zero since all side effects are backed up).

\

[End of Chapter 10]

Title: VAX-11 Assembler Notation -- Rev 5

Specification Status: Fully approved

Architectural Status: under ECO control

File: SRBR4.RNO

PDM #: not used

Date: 31-Oct-78

Superseded Specs:

Author: W. Strecker

Typist: L. Principe

Reviewer(s): P.Conklin, D.Cutler, D.Hustvedt, J.Leonard, P.Lipman, D.Rodgers, S.Rothman, B.Stewart, B.Strecker

Abstract: Appendix B gives the assembler notation sufficient to understand the examples in other sections of the System Reference Manual. It also details the notation used to express addressing modes. It includes a full list of all the notations which can be used in an operand and which addressing mode each results in.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Distributed	Strecker	25-Sep-75
Rev 2	ECOs 1-11	Strecker	9-Mar-76
Rev 3	ECOs 12-18 and April Meeting	Conklin	13-Jun-76
Rev 4	Typos	Conklin	21-Mar-77
Rev 5	Updates	Bhandarkar	31-Oct-78

Rev 4 to Rev 5:

1. Update register mode for 128-bit operands

Rev 3 to Rev 4:

1. Typos
2. D(Rn), D(Rn)[Rx] generate register deferred if D=0.
3. Default D is word, G is longword.

Rev 2 to Rev 3:

1. Change UNDEFINED to UNPREDICTABLE
2. Add B^#, W^#, L^#, Q^# notations
3. Add @B^D(R), @W^D(R), @L^D(R) to summary
4. Change S^@D(AP) to @S^D(AP)
5. Add absolute addressing
6. Add 6-byte addressing
7. Correct typos
8. Remove special <..> notation
9. Correct typos
10. Add assembler/linker checks of displacement, local, and argument modes.
11. Update to Chapter 3 Rev 4 Addressing Modes
12. Change immediate to I prefix
13. Add @R notation
14. Add assembler ban of (PC)
15. @(R) was missing

Rev 1 to Rev 2:

1. Remove R-R mode
2. Introduce branch displacements
3. Add index mode

4. Add summary

5. Add special notation

[End of SRBR4.RNO]

APPENDIX B

ASSEMBLER NOTATION

31-Oct-78 -- Rev 5

B.1 INTRODUCTION

The VAX-11 assembler provides, as a subset, a notation which is very similar to the PDP-11 assembler notation. The principal differences are due to the fact that the VAX-11 architecture has new addressing modes and has several length variations of modes for which the PDP-11 has only a single length. For example, the PDP-11 has displacement addressing with a single displacement size of 16 bits. VAX-11 has displacement addressing in various forms with displacements of 8, 16, and 32 bits.

In general, the programmer need not be aware of the length variations in VAX-11 addresssing modes. The programmer simply writes the addressing mode in a format identical to the analogous PDP-11 addressing mode, and the assembler will choose the shortest form of addressing consistent with the state of symbol definition at assembly time. Occasionally, a programmer may wish to force a given length addressing mode. The VAX-11 assembler includes a notation for accomplishing this. (Of course, if the programmer forces a length which cannot be accommodated at assembly or link time, the assembler or linker will generate an error indication.)

B.2 NOTATION FOR GENERAL MODE ADDRESSING

B.2.1 Register Mode

The general notation is Rn. Since results are UNPREDICTABLE if R is PC for operands taking a single register, or if Rn is SP or PC for operands taking a pair of registers, or if Rn is AP, FP, SP, or PC for operands taking four registers, the assembler generates an error indication.

B.2.2 Register Deferred Mode

The general notation is (Rn). Since results are UNPREDICTABLE if Rn is PC, the assembler generates an error indication.

B.2.3 Autoincrement Mode

The general notation is (Rn)+. For immediate mode see B.2.8.

B.2.4 Autoincrement Deferred Mode

The general notation is @(Rn)+. For absolute addressing mode see B.2.9.

B.2.5 Autodecrement Mode

The general notation is -(Rn). Since results are UNDEFINED if R is PC the assembler generates an error indication.

B.2.6 Displacement Mode

The general notation is D(Rn). To force a byte, word, or long displacement, the notation is B^D(Rn), W^D(Rn), L^D(Rn) respectively. If a general address G is used, the assembler assembles this as D(PC) where D = G - (updated value of PC). This latter form is termed PC-relative addressing. If a form is forced which is shorter than the actually needed displacement, the assembler or linker generates an error indication.

B.2.7 Displacement Deferred Mode

The general notation is @D(Rn). To force a byte, word, or long displacement, the notation is @B^D(Rn), @W^D(Rn), @L^D(Rn) respectively. If a general address @G is used, the assembler assembles this as @D(PC) where D = G - (updated value of PC). This latter form is termed PC-relative deferred addressing. If a form is forced which is shorter than the actually needed displacement, the assembler or linker generates an error indication.

B.2.8 Literal Mode

The general notation is #cons. This results in, depending on the value of the cons, either immediate or literal mode. To force literal mode, the notation is S^#cons. To force immediate mode, the notation is I^#cons. If either literal or immediate mode is used on a modify or write operand, the assembler generates an error indication.

B.2.9 Absolute Addressing Mode

To force a reference to an absolute address the notation is @#location. This is assembled as autoincrement deferred using PC.

B.2.10 General Addressing

When a reference to a symbol will be either absolute or PC-relative, but the choice is to be determined by the linker, the notation is G^location. This is assembled as a five byte operand. The linker chooses either @#location or L^D(PC) depending on whether the location is absolute or PC-relative. This is used both for general external references and for general references between program sections (PSECTs).

B.2.11 Index Mode

The general notation is <base operand mode>[Rx] where <base operand mode> is the notation for any of the addressing modes register deferred, autoincrement (immediate), autoincrement deferred (absolute), autodecrement, displacement (PC-relative), displacement deferred (PC-relative deferred) or general addressing. Since the result is UNPREDICTABLE if a register in the base operand mode is the same as the index register (except for PC), the assembler generates an error.

B.3 GENERAL MODE ADDRESSING SUMMARY

Symbolic	Assembled Mode
1. R	register
2. (R)	register deferred
3. (R)+	autoincrement

4.	-(R)	autodecrement
5.	D(R)	byte, word, or longword displacement. register deferred. default is word if D is not known
6.	B^D(R)	byte displacement
7.	W^D(R)	word displacement
8.	L^D(R)	longword displacement
9.	G	byte, word, or longword displacement off PC default is longword if G is not known
10.	B^G	byte displacement off PC
11.	W^G	word displacement off PC
12.	L^G	longword displacement off PC
13.	G^G	general addressing (absolute or PC-relative)
14.	#cons	autoincrement of PC (immediate) or literal
15.	S^#cons	short literal
16.	I^#cons	immediate
17.	(R)[Rx]	register deferred indexed
18.	(R)+[Rx]	autoincrement indexed
19.	#cons[Rx]	autoincrement of PC (immediate) indexed \This is probably not useful.\
20.	I^#cons[Rx]	autoincrement of PC (immediate) indexed \This is probably not useful.\
21.	-(R)[Rx]	autodecrement indexed
22.	D(R)[Rx]	byte, word, or longword displacement indexed. register deferred indexed.
23.	B^D(R)[Rx]	byte displacement indexed
24.	W^D(R)[Rx]	word displacement indexed
25.	L^D(R)[Rx]	longword displacement indexed

26.	G[Rx]	byte, word, or longword displacement off PC indexed
27.	B^G[Rx]	byte displacement off PC indexed
28.	W^G[Rx]	word displacement off PC indexed
29.	L^G[Rx]	longword displacement off PC indexed
30.	G^location[Rx]	general (absolute or PC-relative) indexed
31.	@(R)[Rx]	byte displacement deferred indexed with 0 displacement
32.	@(R)+[Rx]	autoincrement deferred indexed
33.	@#location[Rx]	autoincrement of PC (immediate) deferred indexed
34.	@D(R)[Rx]	byte, word, or longword displacement deferred indexed.
35.	@B^D(R)[Rx]	byte displacement deferred indexed
36.	@W^D(R)[Rx]	word displacement deferred indexed
37.	@L^D(R)[Rx]	longword displacement deferred indexed
38.	@G[Rx]	byte, word, or longword displacement off PC deferred indexed
39.	@B^G[Rx]	byte displacement off PC deferred indexed
40.	@W^G[Rx]	word displacement off PC deferred indexed
41.	@L^G[Rx]	longword displacement off PC deferred indexed
42.	@(R)	byte displacement deferred with 0 displacement
43.	@(R)+	autoincrement deferred
44.	@#location	autoincrement of PC (immediate)
45.	@D(R)	byte, word, longword displacement deferred
46.	@B^D(R)	byte displacement deferred
47.	@W^D(R)	word displacement deferred
48.	@L^D(R)	longword displacement deferred

49.	@G	byte, word, or longword displacement off PC deferred
50.	@B^G	byte displacement off PC deferred
51.	@W^G	word displacement off PC deferred
52.	@L^G	longword displacement off PC deferred

B.4 BRANCH DISPLACEMENT ADDRESSING

The general notation is locn, where locn is the branch address. The assembler fills in the displacement displ where displ = locn - {updated value of PC}.

B.5 GENERIC OPCODE SELECTION

As a convenience to the programmer, the assembler automatically selects from among similar instructions. This allows the programmer to write code without worrying about these distinctions.

B.5.1 Branch Selection

If the programmer gives BR or BSB as the mnemonic, the assembler will automatically select either BRB or BRW (BSBB or BSBW) based on the distance to the label. If the label is not yet defined, the word branch displacement form will be selected.

B.5.2 Number Of Operand Selection

If the programmer omits the final digit from those opcodes which have two forms (e.g., ADDW instead of ADDW2 or ADDW3), the assembler will select the correct form based on the number of operands specified by the user.

\Initially, the assembler always chooses the shorter form.\

\The idea of selecting the correct data type was rejected because the assembler has no notion of data type. The idea of doing jump/branch resolution (including reversing the test condition) was rejected as better left to compilers.\

[End of Appendix B]

Title: VAX-11 Procedure Calling Specification -- Rev 5

Specification Status: Pending VAXA Approval

Architectural Status: Under ECO control

File: SRCR5.RNO

PDM #: not used

Date: 1-Feb-79

Superseded Specs: Rev 4

Author: Richard Grove

Typist: Gerry Hesley

Reviewers: R. Brender, P. Conklin, D. Cutler, L. Frampton, A. Goldstein, T. Hastings, D. Hustvedt, H. Jacobs, P. Lipman, T. Rarich, R. Shaw, M. Spier

Abstract: Appendix C contains a specification for use of the VAX-11 hardware CALL mechanism by standard higher level languages, BLISS, and the assembly language. This mechanism will be used as the inter-module CALL interface for all major VAX-11 subsystems, including: RMS and the VAX/VMS Operating System.

Specifications for calling sequences, argument transmission, stack usage, and state preservation are included.

Revision History:

Rev.#	Description	Author	Revised Date
Rev 1	Original specification	R. Grove	6-Feb-76
Rev 2	Completion of specification	R. Grove	5-Mar-76
Rev 3	Update to SRM Rev 3 and format	P. Conklin	13-Jun-76
Rev 4	ECOs	P. Conklin	30-Mar-77
Rev 5	ECOs	D. Bhandarkar/ T. Hastings	1-Feb-79

Rev 4 to Rev 5

1. Octaword, G_floating, H_floating.
2. ASCII text string, right justified data type.
3. Decimal Scalar String Descriptor.

Rev 3 to Rev 4:

1. Packed decimal ECO.
2. CF to FP ECO.
3. Remove \ comments including rejected alternative of self-describing arglists.
4. Mask frame<28> reserved to DEC.
5. Add DSC$ mnemonics.
6. Add data types 22, 23.
7. Add classes 0, 2, 6 to 8. Change 2 to 3 and 3 to 5.
8. Move bounds block. Remove virtual array.
9. Note that UNWIND alters return point.
10. Note that procedures may or may not handle omitted trailing null arguments.
11. Clarify that R0 and R1 can not contain two quantities.
12. Note that the stack below (SP) belongs to interrupt and exception routines.
13. Document that complex has real before imaginary.
14. Strings, arrays, and procedures can not be R0/R1 function values.
15. If string function, then push args over one.

Rev 2 to Rev 3:

1. Bring into SRM format as an appendix
2. Convert from pointers to 32-bit addresses
3. Reverse address and length in descriptors

4. Add zoned numeric string format

5. Explain why entry mask must be used

6. Reserve classes and types to CSS

7. Note interchangability of varying and fixed strings for reading

8. Length of packed string is in digits (nibbles-1)

9. Change name to DSC_FL_COLUMN and DSC_POINTER

10. Swapped multipliers and limits in array descriptor

Rev 1 to Rev 2:

1. Reverse address and length in description

2. Change register order in call frame

3. Arg count is unsigned byte

[End of SRCR5.RNO]

APPENDIX C

PROCEDURE CALLING STANDARD

1-Feb-79 -- Rev 5

C.1 INTRODUCTION

This appendix specifies a software standard for use of the VAX-11 hardware procedure CALL mechanism. This standard is applicable to all externally CALLable interfaces in DEC-supported standard system software.

This standard applies to all external procedure CALLs generated by standard DEC language processors, including:

BASIC+2/VAX
BLISS/VAX
COBOL/VAX
FORTRAN IV-PLUS/VAX

This standard is also applicable to inter-module CALLs to major VAX-11 subsystems.

This standard does not apply to calls to internal (or local) routines. Within a single module, the language processor or programmer may use a variety of other linkage and argument-passing techniques.

This standard specifies a single mechanism to be used in a wide variety of applications. As a result, certain compromises have been made between the conflicting requirements of generality, uniformity, and efficiency. Thus:

1. The standard defines and supports the use of call-by-value, call-by-reference, and call-by-descriptor. This permits the designer of a procedure to make trade-offs between generality and efficiency. The cost of this freedom is a somewhat more complex interface as perceived by the caller of a procedure.

2. The procedure CALL mechanism depends on agreement between the calling and called procedures to interpret the argument list. The argument list itself is not fully self-describing.

C.2 GOALS AND NON-GOALS

Goals for the VAX-11 procedure CALLing standard are:

1. The standard must be applicable to all of the inter-module CALLable interfaces in the VAX-11 software system. Specifically, the standard must consider the requirements of BASIC, COBOL, FORTRAN, BLISS, MARS and CALLs to the operating system. Thus:

    1. The standard must support all of the calling capabilities needed for the higher-level languages which DEC now supports (BASIC, COBOL, FORTRAN).

    2. The needs of other languages which DEC may support in the future must be noted (PL/1, Algol, APL).

    3. It must be possible to write calling and called procedures in BLISS and MARS which conform to the standard.

    4. The standard should not include capabilities for lower-level components (e.g., BLISS, MARS, operating system) which cannot be invoked from the higher-level languages.

2. The calling program and called procedure can be written in different languages, including any of the above.

3. The procedure mechanism must be sufficiently economical in both space and time to be used and usable as the only calling mechanism within VAX-11.

4. The standard should contribute to the writing of error-free, modular, and maintainable software. Effective sharing and re-use of VAX-11 software modules is a significant goal.

5. The standard must allow the called procedure a variety of techniques for argument handling. The called procedure may (1) reference arguments indirectly through the argument list, (2) copy scalars and array addresses, (3) copy addresses of scalars and arrays.

Some possible attributes of a procedure-calling mechanism have been considered and rejected. Specific non-goals for the VAX-11 procedure CALL mechanism include:

1. It is not necessary for the procedure mechanism to provide complete checking of argument data types, data structures, and parameter access. The VAX-11 protection and memory-management system is not dependent upon "correct" interactions between user-level calling and called procedures. Such extended checking may be desirable in some circumstances, but system integrity is not dependent upon it.

2. The VAX-11 procedure mechanism need not provide complete information for an interpretive DEBUG facility. The definition of the DEBUG facility will include a DEBUG symbol table which contains the required descriptive information.

C.3 DEFINITIONS

Procedure - A procedure is a routine which follows this specification. A procedure may return values via the argument list and/or the standard value return registers.

Function - A function is a procedure which returns a single value in the value registers. If additional values are returned, they are returned via the argument list.

Subroutine - A subroutine is a procedure which does not return a known value in the value registers. If values are returned, they are returned via the argument list.

Address - A 32-bit VAX-11 address positioned in a longword item.

Exception Condition - Any procedure may be defined to have cases for which it will fail or produce an exception condition. Such failures are indicated by:

1. Returning a function value indicating failure, or

2. SIGNALing a failure using the standard VAX-11 exception condition mechanism, see Appendix D.

The choice of exception condition reporting mechanism is a part of the interface specification for each procedure.

OTS - Object Time System. A collection of procedures which help support a higher level language. OTS procedures are called using the standard CALL mechanism.

C.4 CALLING SEQUENCE

The calling program invokes the called procedure using either the CALLG or CALLS instruction:

```
        CALLG   Arglist.ab, Proc.ab
or
        CALLS   Argcnt.rl, Proc.ab
```

CALLS pushes the argument count Argcnt.rl onto the stack as a longword and sets AP to the top of the stack. The complete sequence using CALLS is thus:

```
        Push    Argn
        ...
        Push    Arg1
        CALLS   #n,Proc
```

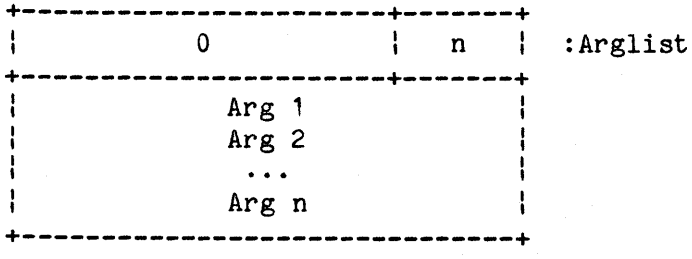
If the called procedure returns control to the calling procedure, control must return to the instruction immediately following the CALLG or CALLS instruction. Skip returns and GOTO returns are prohibited except during UNWIND, see appendix D.

The called procedure returns control to the calling procedure by executing the return instruction, RET.

C.5 ARGUMENT LISTS

C.5.1 Argument List Format

The format of the argument list is a sequence of longwords:



The argument count n is an unsigned byte contained in the first byte of the list. The high order 24 bits of the first longword are reserved for future use and must be zero. To access the argument count, the called procedure must ignore the reserved bits and pick up the count with the equivalent of a MOVZBL instruction.

Each Arg entry in the argument list is a 32-bit longword value. These 32-bit values may be:

1. An uninterpreted 32-bit value.

2. An address; typically a pointer to a scalar data item, an array, or a procedure.

3. An address of a descriptor; descriptors are discussed more fully below.

The standard thus permits simple call-by-value, call-by-reference, call-by-descriptor, or combinations of these. Interpretation of each argument list entry depends upon agreement between the calling and called procedures.

A procedure having no arguments is CALLed with a list consisting of a 0 argument count longword. This is efficiently accomplished by

```
CALLS   #0, Proc.
```

A missing or null argument, for example CALL SUB(A,,B), is represented on VAX-11 by an Arglist entry consisting of a longword 0. Some procedures allow trailing null arguments to be omitted, others require all arguments; refer to the procedure's specification for details.

The argument list must be treated as read-only data by the called procedure.

C.5.2 Argument Lists And Higher-level Languages

Higher-level language functional notations for procedure CALLs are mapped into VAX-11 argument lists according to the following rules:

1. Actual arguments are mapped left-to-right to increasing argument list offsets. The left-most (first) actual argument corresponds to Arglist +4, the next to Arglist +8, etc.

2. Each actual argument position corresponds to a single VAX-11 argument list entry.

C.5.2.1 Order Of Actual Argument Evaluation - Since most higher-level languages do not specify the order of evaluation (with respect to side effects) of actual arguments, those language processors may evaluate actual arguments in any convenient order.

In constructing an argument list on the stack, a language processor may evaluate arguments right-to-left and push their values on the stack. If call-by-reference is used, actual argument expressions can be evaluated

left-to-right, with pointers to the expression values being pushed right-to-left.

The choice of argument evaluation order and code generation strategy is constrained only by the definition of the particular language. Programs should not be written which depend on the order of evaluation of actual arguments.

C.5.2.2 Language Extensions For Argument Transmission - The VAX-11 procedure standard permits arguments to be transmitted by value, by reference, or by descriptor. Each language processor will have a default set of argument mechanisms. Thus FORTRAN will pass scalars, arrays and functions by reference, and will pass strings (CHARACTER) by descriptor. BASIC, however, will transmit both strings and arrays by descriptor.

A set of language extensions is defined to reconcile the different argument transmission techniques. Each language will be extended to provide the user explicit control of argument transmission in the calling procedure.

An alternative to language extension is to use the (worst) general-case solution in all circumstances. This achieves uniformity at a substantial penalty in space and speed.

Each language must be augmented to provide, and each language processor must support, the following compile-time intrinsic functions:

%VAL(arg) - Corresponding argument list entry is the actual 32-bit value of the argument arg, as defined in the language.

%REF(arg) - Corresponding argument list entry is a pointer to the value of the argument arg, as defined in the language.

%DESCR(arg) - Corresponding argument list entry is a pointer to a VAX-11 descriptor of the argument, as defined in this appendix and the language.

These intrinsic functions can be used in the syntax of a procedure CALL to control generation of the actual argument list. For example:

```
CALL SUB1(%VAL(123), %REF(X), %DESCR(A))
```

The intrinsic functions are a necessary escape mechanism in permitting any procedure to be called by programs written in any higher-level language. Careful design of procedure packages will minimize the actual need to use these escape mechanisms.

C.6 FUNCTION VALUE RETURN

A function value is returned in register R0 if representable in 32 bits and registers R0 and R1 if representable in 64 bits. Two separate 32-bit entities cannot be returned in R0 and R1 because higher level languages could not deal with them. If the function value cannot be represented in 64 bits, the source language list of arguments and formals is shifted by one and the first formal in the argument list is a descriptor for the function value. One of the following mechanisms is used to return the function value:

1. If the maximum length of the function value is known (e.g. octaword and H_floating), the calling procedure can allocate the required storage and pass a pointer to the function value storage as the first argument.

   This method is adequate for CHARACTER functions in FORTRAN and VARYING strings in PL/1.

2. The calling procedure can allocate a dynamic string descriptor. The called procedure then allocates storage for the function value and updates the contents of the dynamic string descriptor. This method requires a heap (non-stack) storage management mechanism.

Procedures, such as operating system CALLs, return a success/fail value as a longword function value in R0. Success returns have bit 0 of the returned value set (Boolean true); failure returns have bit 0 clear (Boolean false). The remaining 31 bits of the value are used to encode the particular success or failure status, refer to Appendix D.

C.7 REGISTER USAGE

The following registers have defined uses:

PC - program counter

SP - stack pointer

FP - current stack frame pointer. Must always point at current frame; no modification permitted within a procedure body.

AP - At the instant of CALL, AP must point to a valid argument list. A parameterless procedure points to an argument list consisting of a single longword containing the value 0.

R0,R1 - Function value return registers. These registers are not preserved by any called procedure. They are available as "free temporaries" to any called procedure.

All other registers (R2, R3,..., R10, R11, and AP, FP, SP, PC) are preserved across procedure calls. The called procedure may use any of these provided that it saves and restores them using the procedure entry mask mechanism. The entry mask mechanism must be used so that any stack unwinding done by the condition handling mechanism will correctly restore all registers.

C.8 STACK USAGE

The stack frame created by the CALLG/CALLS instructions for the called procedure is:

```
on-condition longword (0)   :(SP):(FP)
mask/PSW
AP
FP
PC
R0   (optional)
...
R11   (optional)
```

FP always points at the on-condition longword of the stack frame. Any other use of FP at any time within a procedure is prohibited.

The contents of the stack located at higher addresses than the mask/PSW longword belongs to the calling program; it should not be read or written by the called procedure, except as specified in the argument list. The contents of the stack located at lower addresses than (SP) belongs to interrupt and exception routines; it must be assumed to be garbaged continually.

Local storage is allocated by the called procedure by subtracting the required number of bytes from the SP provided on entry. This local storage is automatically freed by the RET instruction.

Bit 28 of the mask/PSW longword is reserved to DEC for future extensions to the stack frame.

C.9 ARGUMENT DATA TYPES

The following encoding is used for atomic data elements:

Mnemonic	Code	Description
DSC$K_DTYPE_Z	0	Unspecified. The calling program has specified no data type; the called procedure should assume the argument is of the correct type.
DSC$K_DTYPE_V	1	Bit. Ordinarily a bit string; see discussion of descriptors.
DSC$K_DTYPE_BU	2	Byte Logical. 8-bit unsigned quantity. See note at the end of this section.
DSC$K_DTYPE_WU	3	Word Logical. 16-bit unsigned quantity. See note at the end of this section.
DSC$K_DTYPE_LU	4	Longword Logical. 32-bit unsigned quantity. See note at the end of this section.
DSC$K_DTYPE_QU	5	Quadword Logical. 64-bit unsigned quantity. See note at the end of this section.
DSC$K_DTYPE_OU	25	Octaword Logical. 128-bit unsigned quantity. See note at the end of this section.
DSC$K_DTYPE_B	6	Byte Integer. 8-bit signed 2's-complement integer.
DSC$K_DTYPE_W	7	Word Integer. 16-bit signed 2's-complement integer.
DSC$K_DTYPE_L	8	Longword Integer. 32-bit signed 2's-complement integer.
DSC$K_DTYPE_Q	9	Quadword Integer. 64-bit signed 2's-complement integer.
DSC$K_DTYPE_O	26	Octaword Integer. 128-bit signed 2's-complement integer.
DSC$K_DTYPE_F	10	Single-precision Floating. 32-bit VAX-11 F_floating point.
DSC$K_DTYPE_D	11	Double-precision D_Floating. 64-bit VAX-11 D_floating point.
DSC$K_DTYPE_G	27	Double-precision G_Floating. 64-bit VAX-11 G_floating point.
DSC$K_DTYPE_H	28	Quadruple-precision Floating. 128-bit VAX-11

H_floating point.

DSC$K_DTYPE_FC 12 Complex. Ordered pair of single-precision F_floating quantities, representing a complex number. The lower addressed quantity represents the real part, the higher addressed represents the imaginary part.

DSC$K_DTYPE_DC 13 Double-precision Complex. Ordered pair of double-precision D_floating point quantities, representing a complex number. The lower addressed quantity represents the real part, the higher addressed represents the imaginary part.

DSC$K_DTYPE_GC 29 Double-precision Complex. Ordered pair of double-precision G_floating point quantities, representing a complex number. The lower addressed quantity represents the real part, the higher addressed represents the imaginary part.

DSC$K_DTYPE_HC 30 Quadruple-precision Complex. Ordered pair of quadruple-precision H_floating point quantities, representing a complex number. The lower addressed quantity represents the real part, the higher addressed represents the imaginary part.

The following string types are ordinarily described by a string descriptor. The data type codes below occur in those descriptors:

DSC$K_DTYPE_T 14 ASCII text string. A sequence of 8-bit ASCII characters.

DSC$K_DTYPE_NU 15 Numeric string, unsigned. See note at the end of this section.

DSC$K_DTYPE_NL 16 Numeric string, left separate sign.

DSC$K_DTYPE_NLO 17 Numeric string, left overpunched sign.

DSC$K_DTYPE_NR 18 Numeric string, right separate sign.

DSC$K_DTYPE_NRO 19 Numeric string, right overpunched sign.

DSC$K_DTYPE_NZ 20 Numeric string, zoned sign.

DSC$K_DTYPE_P 21 Packed decimal string.

DSC$K_DTYPE_ZI 22 Sequence of instructions.

DSC$K_DTYPE_ZEM 23 Procedure entry mask.

The following type codes are RESERVED for future use:

24	RESERVED to DEC
31-191	RESERVED to DEC
192-255	RESERVED to CSS and customers

Note:

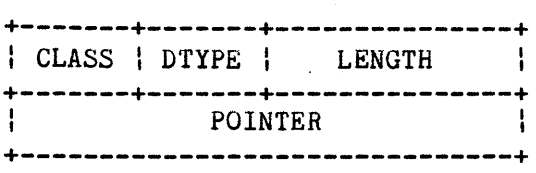
The unsigned data types (codes 2,3,4,5,15,25) do not allocate any space for representing the sign. All bit or character positions are used to represent significant data.

C.10 ARGUMENT DESCRIPTORS

A uniform descriptor mechanism is defined for use by all procedures which conform to this standard. Descriptors are uniformly typed and the mechanism is extensible. As new varieties of descriptor become necessary, they will be added to this catalogue.

C.10.1 Descriptor Prototype

Each class of descriptor consists of at least 2 longwords in the following format:

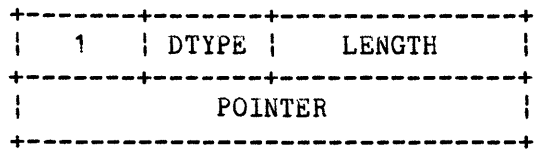


DSC$W_LENGTH <0,15:0>	A one-word field specific to the descriptor class; typically a 16-bit (unsigned) length.
DSC$B_DTYPE <0,23:16>	A one-byte atomic data type code (see C.9)
DSC$B_CLASS <0,31:24>	A one-byte descriptor class code (see below)
DSC$A_POINTER <1,31:0>	A longword pointing to the first byte of the data element described.

Note that the descriptor can be placed in a pair of registers with a MOVQ instruction and then the length and address used directly. This gives a word length, so the class and type are placed as bytes in the rest of that longword. Class 0 is unspecified and hence no more than the above information can be assumed.

C.10.2 Scalar, String Descriptor (DSC$K_CLASS_S)

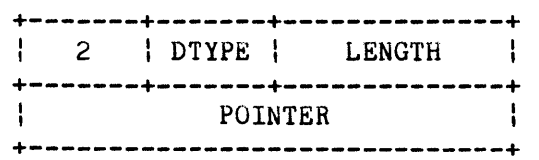
A single descriptor form is used for scalar data and simple strings.



DSC$W_LENGTH	Length of data item in bytes, unless DTYPE EQLU 1 (Bit), or 21 (Packed Decimal). Length of data item is in bits for bit string. Length of data item is in digits (nibbles-1) for packed string.
DSC$B_DTYPE	
DSC$B_CLASS	1=DSC$K_CLASS_S
DSC$A_POINTER	Address of first byte of data storage

C.10.3 Dynamic String Descriptor (DSC$K_CLASS_D)

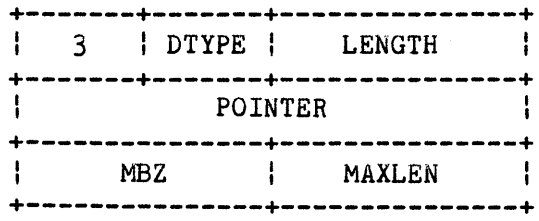
The following descriptor form is used for variable length strings that do not have a fixed maximum length (e.g., BASIC strings). When the string is written, both the length and address fields may be modified. Space is presumed to be allocated dynamically. Note that for reading, this format is interchangeable with type 1 (fixed string).



DSC$W_LENGTH	Current length of data item (in bytes, unless DTYPE EQLU 1, or 21).
DSC$B_DTYPE	
DSC$B_CLASS	2=DSC$K_CLASS_D
DSC$A_POINTER	Current address of lowest addressed byte of string.

C.10.4 Varying String Descriptor (DSC$K_CLASS_V)

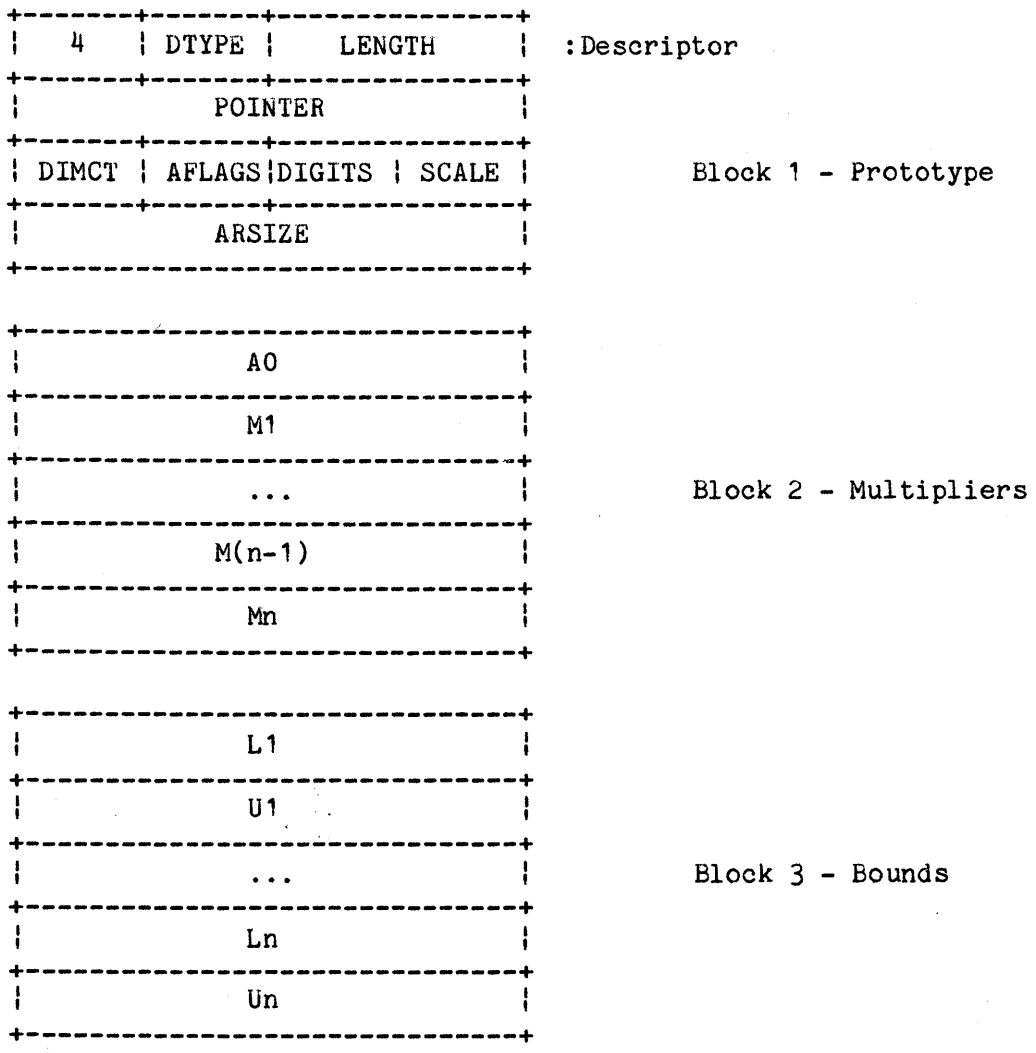
The following descriptor form is used for variable length strings having a fixed maximum length (e.g., PL/1 VARYING). Note that for reading, this format is interchangeable with type 1 (fixed string).



DSC$W_LENGTH	Current length of data item (in bytes unless DTYPE EQLU 1, or 21).
DSC$B_DTYPE	
DSC$B_CLASS	3=DSC$K_CLASS_V
DSC$A_POINTER	Address of lowest addressed byte of string.
DSC$W_MAXLEN	Maximum length of data item (in bytes
<2,15:0>	unless DTYPE EQLU 1 or 21).
<2,31:16>	MBZ; reserved to DEC.

C.10.5 Array Descriptor (DSC$K_CLASS_A)

An array descriptor consists of 3 contiguous blocks. The first block contains the descriptor prototype information and is part of every array descriptor. The second and third blocks are optional. If the third block is present then so is the second. A complete array descriptor has the form:



DSC$W_LENGTH	Data element size (in bytes unless DTYPE EQLU 1, or 21)
DSC$B_DTYPE	
DSC$B_CLASS	4=DSC$K_CLASS_A
DSC$A_POINTER	Address of first actual byte of data storage.
DSC$B_SCALE	Signed power of ten multiplier to convert the internal form to external form. For example, if internal number is 123 and scale is +1, then the represented external number is 1230.
DSC$B_DIGITS	If non-zero, unsigned number of decimal digits

in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH.

DSC$B_AFLAGS <2,23:16> — Array flag bits.

- Reserved <2,20:16> — MBZ
- DSC$V_FL_COLUMN <2,21> — If set, the elements of the array are stored by columns (FORTRAN). Otherwise the elements are stored by rows.
- DSC$V_FL_COEFF <2,22> — If set, the multiplicative coefficients in Block 2 are present.
- DSC$V_FL_BOUNDS <2,23> — If set, the bounds information in Block 3 is present.

DSC$B_DIMCT <2,31:24> — Number of dimensions

DSC$L_ARSIZE <3,31:0> — Total size of array (in bytes unless DTYPE EQLU 1 or 21).

DSC$A_A0 <4,31:0> — Address of element A(0,0,...,0). This need not be within the actual array; it is the same as DSC$A_POINTER for 0-origin arrays.

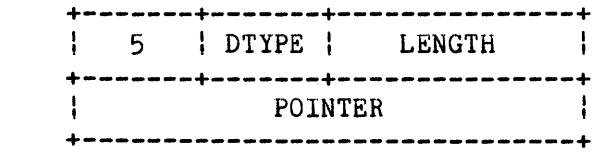
DSC$L_Mi <4+i,31:0> — Addressing coefficients (Mi = Ui-Li+1)

DSC$L_Li <3+n+2*i,31:0> — Lower bound of i'th dimension.

DSC$L_Ui <4+n+2*i,31:0> — Upper bound of i'th dimension.

C.10.6 Procedure Descriptor (DSC$K_CLASS_P)

The descriptor for a procedure specifies its entry address and function value data type, if any.



DSC$W_LENGTH	Length associated with the function value.
DSC$B_DTYPE	Function value data type.
DSC$B_CLASS	5=DSK$K_CLASS_P
DSC$A_POINTER	Address of entry mask to routine.

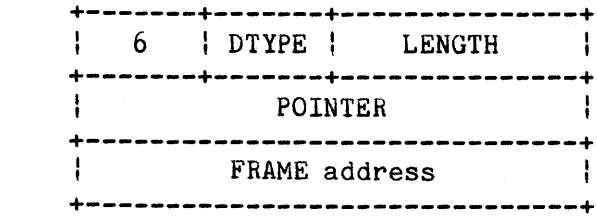
Procedures return values in R0 or R0 and R1 as follows:

1. If a scalar, then the value is in R0 or R0 and R1. The type and length are specified as DSC$B_DTYPE and DSC$W_LENGTH in the procedure descriptor.

2. If not a scalar (i.e., if an array, a string, a procedure, etc.), then no function value may be returned. Instead, the argument expressed as a function value is instead passed as the first argument and the other arguments are shifted down by one.

C.10.7 Procedure Incarnation Descriptor (DSC$K_CLASS_PI)

The descriptor for a procedure incarnation is the same as a procedure descriptor with the addition of its call frame address. This is used to refer to a specific incarnation of a procedure, such as in ALGOL or PL/I.

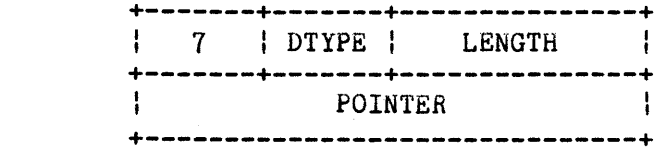


DSC$W_LENGTH	Length associated with the function value
DSC$B_DTYPE	Function value data type.
DSC$B_CLASS	6=DSC$K_CLASS_PI.
DSC$A_POINTER	Address of entry mask to routine.
DSC$A_FRAME	Address of frame of this incarnation.

<2,31:0>

C.10.8 Label Descriptor (DSC$K_CLASS_J)

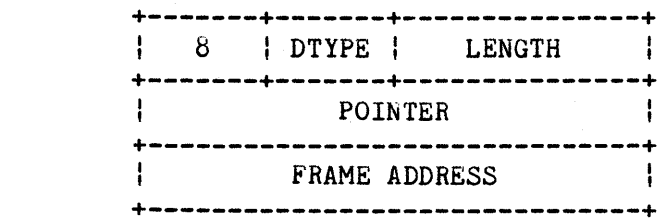
The descriptor for a label specifies the start of its code.



DSC$W_LENGTH Not used; MBZ.
DSC$B_DTYPE Not used; MBZ.
DSC$B_CLASS 7=DSC$K_CLASS_J
DSC$A_POINTER Address of label to jump to.

C.10.9 Label Incarnation Descriptor (DSC$K_CLASS_JI)

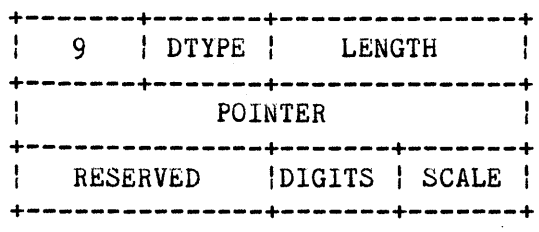
The descriptor for a label incarnation is the same as a label descriptor with the addition of its procedure incarnation's call frame address. This is used to refer to a label within a specific incarnation of a procedure, such as in ALGOL or PL/I.



DSC$W_LENGTH Not used; MBZ.
DSC$B_DTYPE Not used; MBZ.
DSC$B_CLASS 8=DSC$K_CLASS_JI.
DSC$A_POINTER Address of label to jump to
DSC$A_FRAME Address of frame of this incarnation.
<2,31:0>

C.10.10 Decimal Scalar String Descriptor (DSC$K_CLASS_SD)

A single descriptor form is used to give decimal size and scaling information for scalar data and simple strings.



DSC$W_LENGTH	Length of data item in bytes, unless DTYPE EQLU 1 (Bit), or 21 (Packed Decimal). Length of data item is in bits for bit string. Length of data item is in digits (nibbles-1) for packed string.
DSC$B_DTYPE	
DSC$B_CLASS	9=DSC$K_CLASS_SD
DSC$A_POINTER	Address of first byte of data storage
DSC$B_SCALE	Signed power of ten multiplier to convert the internal form to external form. For example, if internal number is 123 and scale is +1, then the represented external number is 1230.
DSC$B_DIGITS	If non-zero, unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH.
Reserved <2,31:16>	Reserved for future use. MBZ.

C.10.11 Reserved Descriptors

Descriptor classes 10-191 are RESERVED to DEC. Classes 192 through 255 are RESERVED to CSS and customers. \Descriptor class 10 has been assigned to COBOL-74 and will be documented later.\

[End of Appendix C]

Title: VAX-11 Condition Handling Facility -- Rev 5

Specification Status: Approved

Architectural Status: Under Change Control

File: SRDR5.RNO

PDM #: not used

Date: 31-Mar-77

Superseded Specs:

Author: P. Conklin, T. Hastings

Typist: P. Conklin

Reviewer(s): R. Blair, R. Brender, P. Conklin, D. Cutler, R. Grove, D. MacLaren, J. Rudy, M. Schwartzman

Abstract: Appendix D defines a unified facility which supports the exception handling mechanisms needed by each of the common languages. It provides the programmer with some control over fixup, reporting, and flow of control on errors. It provides subsystem and application writers with the ability to override system messages in order to give a more suitable application oriented interface.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	original	L. Frampton	Mar-76
Rev 2	major expansion	M. Schwartzman	21-Apr-76
Rev 3	general cleanup and proposal	T. Hastings	17-Sep-76
Rev 4	heavy pruning	P. Conklin	12-Nov-76
Rev 5	result of implementation	P. Conklin	31-Mar-77

Rev 4 to Rev 5:

1. Add minimum constraints to goals.
2. Change CHF$ to LIB$ or SYS$ or SS$ where appropriate.
3. Document that hardware exceptions are treated like calls to SIGNAL.
4. Add a 65K stack frame depth limit.
5. Add table of actions.
6. Add SS$_INSFRAME error.
7. Note that a handler may be called back on unwind.
8. Remove table with summary of symbols.
9. Document that severity is condition<2:0>.
10. Add definition of exception vectors to section 4.
11. Change name from SET_UNWIND to UNWIND.
12. Allow handler to modify the signal argument vector. In particular, to modify the severity of condition_value.
13. Add symbols for offsets.

Rev 5 to Rev 4:

1. Remove function vs. subroutine call indicator.
2. Remove ERRRET function and ability to control exception signaling vs. status reporting.
3. Remove AHL and ability to list a set of Handlers.
4. Simplify REVERT.
5. Remove PUSH/POP Handler.
6. Remove special conditions ERRRET and MATCH_ANY.
7. Remove ENABLE/DISABLE except for H/W PSW bits.
8. Remove SIGNAL DISABLEd.
9. Add unwind_PC to Handler args.

10. Replace STARLET resignal argument with frame depth to Handler.

11. Replace establisher args to Handler with establisher frame argument.

12. Add MBZ arg for future growth.

13. Add Handler return value of CHF$E_RESIGNAL.

14. Shift Handler control return value bits left one.

15. Add OTS frame above main program.

16. Make names LEQ 15 characters.

17. Remove CHF$E_REQ_TO_TERMINATE, CHF$E_TERMINATE, and CHF$TERMINATE. They can be added in the future if needed.

18. Remove handler and Signaler look ahead. Move default action to be at top of stack and to always issue message. The error status codes are error conditions and result in image exit.

19. Add vectored Handlers.

20. Modify handler argument format to agree with that from system signals. Solves the R0/R1 problem.

21. On multiple active signals, enter Handler's Handler, not establisher's Handler.

22. Remove arithmetic exceptions since they are a system exception.

23. Change "process termination" to "image exit".

24. Remove Explicit versus Implicit terminology.

25. Change fac$E_... to fac$_...

26. Add procedure library argument notation.

27. Change CF to FP.

28. Change handler arg layout.

29. Add CHF$STOP.

30. Change CHF$UNWIND to CHF$SET_UNWIND with different semantics.

31. Add example to Multiply active signals.

32. Remove BLISS, OTS, and STARLET changes.

33. Remove redundant or obsolete definitions.

Rev 2 to Rev 3:

1. Eliminate conditional Handler.

2. Specify specific routines and codes.

3. Eliminate PROPAGATE-ENABLE/DISABLE.

4. Add PUSH/POP Handler.

5. Add UNWIND.

6. Add ERRRET.

Rev 1 to Rev 2:

1. Add special case of multiple active signals.

2. Establish consistent conceptual framework.

3. Establish consistent terminology.

[End of SRDR5.RNO]

APPENDIX D

CONDITION HANDLING FACILITY

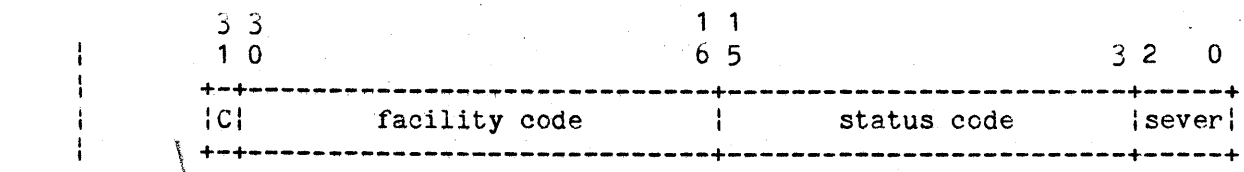
31-Mar-77 -- Rev 5

D.1 TERMINOLOGY

Condition Handling Facility - A collection of VAX-11 Library procedures and operating system services which implement the condition handling mechanisms of the VAX-11 system.

Exception Condition - A condition occurring during an operation which is brought to the attention of the invoker of the operation. Although an exception condition usually represents a failure, it is not restricted to error situations.

Condition Value - A 32-bit quantity which uniquely specifies an exception condition. The format is:



C = 1 if Customer or CSS facility code
0 if Digital facility code

sever is the severity code as follows:
bit <0> is set for success (logical true)
bits <2:1> distinguish degrees of success or failure
thus, the field <2:0> can be considered as a number
0 = warning
1 = success
2 = error
4 = severe_error
3, 5, 6, 7 reserved to DEC for the future

Condition Symbol - A global symbol used to specify a condition value. A condition symbol has the form:

fac$_symbol

where fac is the three letter facility name and symbol is a string of alphanumerics and underlines which identify the specific condition.

Procedure Activation - The environment in which a procedure executes including a unique stack frame allocated in the process stack for temporary storage.

Handler - A function which a procedure activation establishes. A handler is called when an exception condition is signaled.

Signal - The standardized procedure call to the signal procedure to indicate the occurrence of an exception condition. The signal procedure calls a previously established handler. Signals are also generated by the operating system in response to hardware detected exceptions and some software detected exceptions.

D.2 GOALS

1. Provide what the common OTS needs to support the different languages.

2. Provide the programmer with some control over fixing, reporting, and flow of control on exceptions.

3. Provide subsystem and application writers with the ability to override system messages in order to give a more suitable application oriented interface.

At a minimum, the following must be possible:

4. For Fortran, allow routines to signal an OTS routine above the main program that does the ERRSET check.

5. For BASIC, support the ON - GOTO statement.

6. For COBOL, support the USE - FOR statement.

7. Application programmers must be able to intercept all system, OTS, and library messages. The condition handling routine must be able to issue an alternate message. It must also be able to unwind and send control to a recovery point.

From these goals and constraints can be derived the following subordinate goals:

8. Provide a standardized, language-independent way for handling errors (and other exception conditions) which is usable by all procedures written in any language, including FORTRAN, COBOL, and BASIC, as well as MARS and BLISS. This goal is derived from the goal of allowing any language to call procedures written in any other language. The users of this facility are the Common OTS, subsystem writers, and programmers writing in any of these languages.

9. Handle hardware and software detected exception conditions in the same way with the same functionality, including creation of defaults.

10. Provide a mechanism whose ease of use encourages understandable and reliable programming. In particular the caller of a procedure must only be able to affect its execution in well defined ways.

11. Avoid one pitfall of the PL/I on condition mechanism which is:

    PL/I does not provide any way for intervening procedure activations to perform user specified cleanup when a (non-local) GO TO is performed from a handler to the procedure which established it.

12. Add no space or time overhead to procedure calls and returns which do not establish handlers. Minimize time overhead for establishing handlers at the cost of increased time overhead when exceptions occur.

D.3 RETURNING A CONDITION VALUE

The most reliable means for indicating a software detected exception condition occurring in a called procedure is for the called procedure to return a condition value as a function Value and for the caller to check the return value for TRUE or FALSE. TRUE is bit 0 set and FALSE is bit 0 cleared. TRUE means that the requested operation was performed successfully; FALSE means an error condition occurred; in both cases, the rest of the value is a condition value. Thus, most procedures are written as functions, rather than subroutines. If it is necessary to indicate an exceptional situation without returning a value, then generate a CALL to LIB$SIGNAL, refer to section 7, signaling a condition.

D.4 ESTABLISH A CONDITION HANDLER

For the primary purpose of handling hardware detected exceptions, the VAX-11 system supplies a mechanism for the programmer to specify a handler function to be called when an exception occurs. This mechanism may also be used for software detected exceptions.

Each procedure activation has a condition handler potentially attached to it via a longword in its stack frame. Initially, the longword contains 0, indicating no handler. A handler is established by moving the address of the handler's procedure entry point mask to the establisher's stack frame.

In addition, the operating system provides two exception vectors at each access mode. These vectors are available to declare handlers which take precedence over any handlers declared at the procedure level. These are used, for example, to allow a debugger to monitor all exceptions, whether or not handled. Since these handlers do not obey the procedure nesting rules, they should not be used by modular code. Instead, the stack based declaration should be used.

A procedure library entry point is defined to help users of languages which do not provide explicit syntax to reference the stack frame or to declare handlers:

```
old_handler.flu.r = LIB$ESTABLISH (new_handler.flu.rp)
```

The routine is:

```
MOVL    @12(FP),R0          ;get previous handler
MOVAL   @4(AP),@12(FP)      ;set new handler
RET                         ;return to caller
```

D.5 REVERT HANDLER

The revert handler operation deletes the handler associated with the procedure activation. This is done by clearing the handler address in the stack frame.

A procedure library entry point is defined to help users of languages which do not provide explicit syntax to reference the stack frame or to revert handlers:

```
old_handler.flu.r = LIB$REVERT
```

The routine is:

```
MOVL    @12(FP),R0      ;get previous handler
CLRL    @12(FP)         ;clear handler
RET                     ;return to caller
```

D.6 ENABLING/DISABLING CONDITIONS

The purpose of disabling a condition within a procedure activation is to prevent any established handler for the condition from getting control. This is allowed primarily for the hardware enableable exceptions.

Procedure library entry points are defined to help users of languages which do not provide explicit syntax to enable exceptions or to reference the PSW:

```
old_setting.wlu.v = LIB$FLT_UNDER (new_setting.rlu.r)
old_setting.wlu.v = LIB$INT_OVER (new_setting.rlu.r)
old_setting.wlu.v = LIB$DEC_OVER (new_setting.rlu.r)
```

where old_setting and new_setting are Boolean values.

The routine in each case is:

```
EXTV    pos,#1,4(FP),R0 ;get boolean value of old setting
INSV    @4(AP),pos,#1,4(FP) ;set new value in
                        ; saved PSW
RET                     ;return to caller
```

D.7 SIGNAL A CONDITION

The signal operation is the method used for indicating that an exception condition has occurred. It is primarily intended for indicating the occurrence of hardware detected exceptions.

D.7.1 Signal

When a language or user wishes to issue a signal, it calls the standard procedure:

```
CALL LIB$SIGNAL (condition_value.rlu.v, arg_list...)
```

When a language or user wishes to issue a signal and never continue, it calls the standard procedure:

```
CALL LIB$STOP (condition_value.rlu.v, arg_list...)
```

where in both cases condition_value indicates the condition which is being signaled. Condition_value<31:3> indicate the condition and condition_value<2:0> indicate the severity. The severity should be chosen as follows. If the default action should be to issue a message and continue, then the severity should be 1 (i.e., "success"). Otherwise, the choice should be between 0, 2, and 4 according to what severity of error should be reported to the command processor. In the choice between error and severe_error (2 and 4), the norm is severe_error unless a good reason exists to make it error. The arguments arg_list are a series of arguments describing the details of the exception. These are the same arguments used to issue a system message. Note that unlike most CALLs, LIB$SIGNAL preserves R0 and R1 as well as the other registers. This allows a debugger to display the entire state of the process at the time of the exception. It also allows signals to be placed in assembly language code without changing the register usage. This is useful for debugging checks and statistics gathering. Hardware and system service exceptions behave as though they were a call to LIB$SIGNAL.

The signal procedure examines the two exception vectors and then up to 65K previous stack frames. The current and previous stack frames are found by using FP and chaining back through the stack frames using the saved FP in each frame. The exception vectors are a pair of address locations per access mode.

A frame before the call by the system command processor to the main program establishes a catch-all handler which issues system messages. The catch_all handler uses condition_value to call $GETERR to get the message and then uses $FAO and the arg_list to output the message. If the condition_value<0> is set the catch_all handler returns with SS$_CONTINUE, otherwise it calls SYS$EXIT.

The stack search terminates when the old FP is 0 or is not accessible or 65K frames have been examined. If no handler is found, or all handlers returned with a SS$_RESIGNAL, then SIGNAL issues a message that no handler was found and then issues the SIGNALed message via $GETERR and $FAO and then calls SYS$EXIT. This message is not signalled.

If a handler returns SS$_CONTINUE, and LIB$STOP was not called, then control returns to the signaler. Otherwise LIB$STOP issues a message that there was an attempt to continue from a non-continuable exception and calls SYS$EXIT. This message is not signaled.

All combinations of interaction between handler actions, the default handler, the type of signal, and the call to signal or stop are detailed in the following table.

	condition <0>	catch all gets cntrol (default)	handler specifies continue	handler specifies UNWIND	no handler is found (stack bad)
call to: SIGNAL	1	condition message RET	RET	UNWIND	"no handler found" condition message EXIT
	0	condition message EXIT	RET	UNWIND	"no handler found" condition message EXIT
STOP	1	condition message "can't continue" EXIT	"can't continue" EXIT	UNWIND	"no handler found" condition message EXIT
	0	condition message EXIT	"can't continue" EXIT	UNWIND	"no handler found" condition message EXIT

condition message is the standard message for the condition value as retrieved by $GETERR and formatted by $FAO.

"no handler found" is a standard message which indicates that no handler was found (i.e., that the stack is bad). The message distinguishes between no handler (old FP = 0 or too many frames) and access violation (old FP = junk).

"can't continue" is a standard message which indicates an attempt to continue from a call to LIB$STOP. The message includes part of the standard condition message.

D.7.2 Handler

If a condition handler is found on a software detected exception, the handler is called with a argument list consisting of:

```
continue.wlu.v = handler (signal_args.ml.ra, mechanism_args.ml.ra)
```

where each argument is a reference to a longword vector. The first longword of each vector is the number of remaining longwords in the vector. The symbols CHF$L_SIGARGLST (=4) and CHF$L_MCHARGLST (=8) can be used to reference the handler arguments relative to AP.

Signal_args is the arg list from the call to LIB$SIGNAL or LIB$STOP. In particular, the second longword is the condition_value being signaled. Since bits <2:0> of the condition_value indicate severity and do not indicate which condition is being signalled, the handler should examine only condition_value<31:3>. The setting of bits <2:0> vary depending upon the environment. In fact, some handlers may simply change the severity of a condition and resignal. The symbols CHF$L_SIG_ARGS (=0) and CHF$L_SIG_NAME (=4) can be used to reference the elements of the signal vector.

Mechanism_args is a vector of five longwords

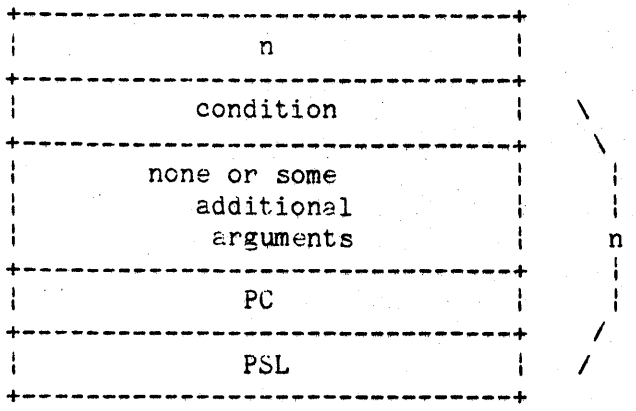
(4.rl.v, frame.rl.v, depth.rl.v, R0.rl.v, R1.rl.v)

4	CHF$L_MCH_ARGS
frame	CHF$L_MCH_FRAME
depth	CHF$L_MCH_DEPTH
R0	CHF$L_MCH_SAVR0
R1	CHF$L_MCH_SAVR1

Frame is the contents of in the establisher's FP. This can be used as a base to reference the local storage of the establisher if the restrictions in section 8 are met. Depth is a positive counter of the number of procedure activation stack frames above the signal that the handler was established. Depth has the value 0 for an exception handled by the activation invoking the exception (i.e., containing the instruction causing the hardware exception or calling LIB$SIGNAL). Depth has positive values for procedure invocations calling the one having the exception (1 for the immediate caller, etc.) If a system service gives an exception, the immediate caller of the service gets notified at depth = 1. Depth has value -2 when the handler is established by the primary exception vector and -1 when it is established by the secondary vector. R0 and R1 are the values of these registers at the time of the call to LIB$SIGNAL.

The handler function must return with a function value to indicate whether it wants the procedure to continue (SS$_CONTINUE) or the signal to be resignaled (SS$_RESIGNAL). If the handler wants to unwind, it calls SYS$UNWIND and then returns. In this case the handler function value is ignored.

For hardware detected exceptions, the condition_value indicates which exception vector was taken and the next 0 or several longwords are the additional parameters as specified in Chapter 6. The remaining two longwords are the PC and PSL:



D.8 OPTIONS OF HANDLER

In order not to impact compiler optimization, a handler and anything it calls is restricted to referencing only explicitly passed arguments. They cannot reference COMMON or other external storage and they cannot reference local storage in the procedure which established the handler. Compilers relaxing this rule must ensure that any variables referenced by the handler are always kept in memory (VOLATILE) and have a full lifetime.

If the handler decides not to handle the condition, it returns to its caller with a function value of SS$_RESIGNAL.

When an exception condition occurs, the handler function performs actions in three distinct areas before returning a function value to the signaler. The handler actions are:

1. Fix the problem:

    1. Take corrective action so that a continue will achieve the desired effect.

    2. SIGNAL a different condition. Note that this SIGNAL will check the exception vectors and then search only frames which were not searched on the previous still active signals. Refer to section 10, Multiple Active Signals.

2. Report the exception:

    1. Keep a count, etc.

    2. RESIGNAL the same condition which will usually report to the user at a terminal or in a log file.

    3. Change the severity field of the condition_value and RESIGNAL.

    4. SIGNAL a different condition.

3. Flow of control:

    1. Continue from the signal. If the signal was issued by a call to LIB$STOP, then LIB$STOP will exit the image.

    2. Unwind to the establisher's caller with establisher function value from R0 and R1 in the mechanism vector.

    3. Unwind to the establisher at the point of the call which resulted in the exception. The callee's function value is taken from R0 and R1 in the mechanism vector.

    4. Unwind to a specified activation and transfer to a specified location with R0 and R1 from the mechanism vector.

D.9 REQUEST TO UNWIND

If the handler decides to unwind, the handler or any procedure it calls performs:

```
success.wlu.v = SYS$UNWIND
                ( [depth.rl.r = {handler depth} + 1],
                  [new_PC.rlu.r = {return PC}] )
```

where depth specifies how many pre-signal frames to remove. If depth is LEQ 0 then nothing is to be unwound. The default is to return from the establisher. To unwind to the establisher, the depth from the call to the handler should be specified. The argument new_PC specifies the location to receive control when the unwind is complete. The function value is a standard success code (SS$_NORMAL), or indicates the failure "no signal active" (SS$_NOSIGNAL), "already unwinding" (SS$_UNWINDING), or "insufficient frames for depth" (SS$_INSFRAME).

The unwind will happen when the handler returns to the condition handling facility. Unwinding is done by scanning back through the stack and calling each handler which has been associated with a frame. The handler is called with exception SS$_UNWIND to perform any application specific cleanup. In particular, if the depth specified includes unwinding the establisher's frame, then the current handler will be re-called with this unwind exception.
The call to the handler is of the same form as described above with the following values:

```
signal_args
        1
        condition_value = SS$_UNWIND

mechanism_args
        4
        frame   establisher's frame
        depth   0 (i.e., unwinding self)
        R0      R0 which unwind will restore
        R1      R1 which unwind will restore
```

After each handler has been called, the stack is cut back to the previous frame.

Note that the exception vectors are not checked because they are not being removed. Any function value from the handler is ignored. In the unlikely case that the handler wants to specify the value of the top level "function" being unwound, it should modify R0 and R1 in the mechanism vector because they will be restored from the mechanism argument vector at the end of the unwind.

D.10 MULTIPLE ACTIVE SIGNALS

A signal is said to be active until the signaler gets control again or is unwound. It is possible for a signal to occur while a handler or a procedure it has called is executing. Consider the following example. For each procedure (A, B, C, ...) let the handler it establishes be (Ah, Bh, Ch, ...). If A calls B calls C which signals "S" and Ch resignals, then Bh gets control. If Bh calls X calls Y which signals "T" the stack is:

```
<signal T>
    Y
    X
    Bh
<signal S>
    C
    B
    A
```

which was programmed:

```
A

  B ------------------> Bh

    C                     X

  <signal S>              Y

                      <signal T>
```

The desired order to search for handlers is Yh, Xh, <Bh>h, Ah. Note that Ch should not be called because it is a structural descendent of B. Bh should not be called again because that would require it to be recursive. If it were recursive, then handlers could not be coded in nonrecursive languages such as FORTRAN. Instead Bh can establish itself or another procedure as its handler (Bhh).

To implement this, the following algorithm is used. As usual, the primary and secondary exception vectors are checked. Then, however, the search backward in the process stack is modified. In effect the stack frames traversed in the first search are skipped over in the second search. Thus, the stack frame preceding the first handler up to and including the frame of the procedure which has established the handler is skipped. Despite this skipping, depth is not incremented. The stack frames traversed in the first and second search are skipped over in a third search, etc. Note that if a handler SIGNALs, it will not automatically be invoked recursively. However, if a handler itself establishes a handler this second handler will be invoked. Thus, a recursive handler should start by establishing itself. Any procedures invoked by the handler are treated in the normal way; that is, exception signaling follows the stack up to the handler.

For proper hierarchical operation, an exception occurring during execution of a handler established in an exception vector should be handled by that handler rather than propagating up the activation stack. This is the vectored handler's responsibility. It is most easily accomplished by the vectored handler establishing a catch-all handler.

D.11 IMPLICATIONS FOR COMPILERS

No compiler giving explicit support for condition handling should allow a handler to access COMMON or external variables because this would impose a constraint on the code generated by other compilers during operations which could trap. For external procedures used as or by handlers, this rule must be observed by the programmer.

[End of Appendix D]

Title: VAX-11 Architectural Subsetting -- Rev 5

Specification Status: Fully approved

Architectural Status: under ECO control

File: SRER5.RNO

PDM #: not used

Date: 6-sep-78

Superseded Specs: Rev 4

Author: T. Hastings

Typist: M. J. Forbes

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Appendix E describes the instructions, and other features of the architecture which may be omitted or provided as a customer option in some processor implementations. These decisions were originally agreed to by the Pruning Committee which consisted of: Bob Armstrong, Jega Arulpragasam, Roger Blair, Ron Brender, Peter Conklin, Dave Cutler, Bruce Delagi, Roger Gourd, Rich Grove, Tom Hastings, Len Hughes, Herb Jacobs, Rich Lary, Dave Rodgers, Steve Rothman, Rich Shaw, Bob Stewart, and Bill Strecker.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Results of pruning meeting	Hastings	18-Dec-75
Rev 2	Made names agree with ECO 5	Hastings	11-Mar-76
Rev 3	ECOs 12-18 and April Meeting	Conklin	27-May-76
Rev 4	ECOs	Conklin	28-Feb-77
Rev 5	G_floating and H_floating	Bhandarkar	6-Sep-78

Rev 4 to Rev 5:

1. Add G_floating.
2. Add H_floating.

Rev 3 to Rev 4:

1. EDITPC ECO.
2. Packed decimal ECO.

Rev 2 to Rev 3:

1. Add MOVF, MOVD to subset
2. Add CVT{B,W,L}{F,D} to subset
3. Add ACBF, ACBD to subset
4. Remove CHOPF, CHOPD, POLYF, POLYD
5. Add CRC as a class
6. Make EDITN a separate class
7. Remove MOVU, MULN4, DIVN4, CVTLU, CVTPU, ASHU
8. Change names to MOVN, MULN, DIVN, CVTLN, CVTPN, ASHN
9. Add MOVTUC, MATCHC to character class
10. Clarify subsetting a class
11. Remove address break
12. Document rules on future additions
13. Note that CRC, MOVTUC, MATCHC are not currently committed for First Machine.
14. Add POLYF, POLYD
15. Make MULN, DIVN a separate class
16. Add SKPC

Rev 1 to Rev 2:

1. Update names per ECO 5

[End of SRER5.RNO]

APPENDIX E

ARCHITECTURAL SUBSETTING

6-Sep-78 -- Rev 5

This appendix describes those parts of the VAX-11 architecture which may be: 1) omitted completely from a processor, or 2) provided to customers as a processor option. The subsetting of the architecture reflects the need for certain agreements between hardware and software implementors in order to be able to trade-off manufacturing cost, software development cost, and performance across all future machines implemented in the VAX-11 family. It is agreed that the first processor will implement all of the instructions in the VAX-11 architecture as defined by April 1976.

These decisions are an attempt to reach a compromise on the following conflicting hardware and software goals for all processors produced in the future in the VAX-11 family:

E.1 GOALS

1. Hardware goal - Permit an implementor of a low end processor to omit instructions and other features in order to reduce manufacturing cost without losing the ability to run all of the system software. The implementor will have some idea of the impact on space and time performance of various classes of software products as he makes his subsetting design decisions.

2. Software goal - Provide as small a number of classes of processor instruction sets as possible to reduce software development costs. In particular a single version of each compiler should run on all processors in the VAX family. Also the combination of hardware and a processor specific operating system kernel will (as required) give the appearance of a complete architecture on all processors.

E.2 DEFINITIONS

Kernel Instruction Set - The kernel instruction set is that subset of the VAX-11 architecture which is always present in all processors in the VAX-11 family.

Subsettable instructions - The subsettable instructions are the instructions in the VAX architecture which processor implementors may choose: (1) to omit completely, (2) to include as a processor option, or (3) to include in the basic processor.

Processor option - An implementation technique whereby a particular processor is designed to provide customers the ability to purchase additional hardware to provide some or all of the omitted instructions or other processor features.

E.3 KERNEL INSTRUCTION SET

The Kernel instruction set includes all of the instructions in the architecture, except the following 6 classes of subsettable instructions:

1. Floating* - MOVF, MNEGF, CVTF{B,W,L}, CVT{B,W,L}F, CMPF, TSTF, ADDF2, ADDF3, SUBF2, SUBF3, MULF2, MULF3, DIVF2, DIVF3, CVTRFL, EMODF, POLYF, ACBF

2. Double* - MOVD, MNEGD, CVTD{B,W,L,F}, CVT{B,W,L,F}D, CMPD, TSTD, ADDD2, ADDD3, SUBD2, SUBD3, MULD2, MULD3, DIVD2, DIVD3, CVTRDL, EMODD, POLYD, ACBD

3. G_floating* - MOVG, MNEGG, CVTG{B,W,L,F}, CVT{B,W,L,F}G, CMPG, TSTG, ADDG2, ADDG3, SUBG2, SUBG3, MULG2, MULG3, DIVG2, DIVG3, CVTRGL, EMODG, POLYG, ACBG

4. H_floating* - MOVH, MNEGH, CVTH{B,W,L,F,D,G}, CVT{B,W,L,F,D,G}H, CMPH, TSTH, ADDH2, ADDH3, SUBH2, SUBH3, MULH2, MULH3, DIVH2, DIVH3, CVTRHL, EMODH, POLYH, ACBH, MOVO, CLRH, MOVAH, PUSHAH

5. Cyclic Redundancy Check - CRC

6. Packed* - MOVP, CMPP3, CMPP4, ADDP4, ADDP6, SUBP4, SUBP6, CVTLP, CVTPL, CVTPT, CVTTP, CVTPS, CVTSP, ASHP

7. Packed Multiply - MULP, DIVP (If packed included and multiply not, then do setup and set FPD).

8. Edit - EDITPC

and except the following class which may become a subsettable part of the VAX architecture:

9. Character* - MOVTC, MOVTUC, CMPC3, CMPC5, SCANC, SPANC, LOCC, SKPC, MATCHC

*Note: All forms of the address data type and all forms of context indexing have been included in the kernel set. In addition, MOVC3 and MOVC5 are part of the kernel set. MOVC3 is the recommended technique for moving blocks of data. Using MOVC5 with a zero length source is the recommended technique for clearing an area of memory.

Each class may only be subset as an entity. This means that if any instruction of a class is included, all instructions of that class must be included.

E.4 GUIDELINES FOR SOFTWARE IMPLEMENTORS

E.4.1 Diagnostic Software

Use only the kernel instruction set.

E.4.2 Operating System Kernel

Use only the kernel instruction set. The operating system kernel (as required) will simulate in software all instructions omitted from a machine. Thus each processor and processor option has a potentially different operating system kernel.

E.4.3 System Software And Compiled Code

This category includes all of the rest of the software, namely compilers, utilities, compiled code, application programs, and operating system support (e.g., file system, data management, etc.). Use the kernel instruction set and the character instructions freely. Use the instructions in a subsettable class only if they are natural for the data being operated upon and are used extensively in the program. The kernel will simulate the missing instructions. Isolated use of subsettable instructions for ad hoc purposes is to be discouraged. For low end machines after the first we will measure the use of the character instructions and determine the performance penalty due to software simulation. No character instructions will be permitted to be subsetted until the measurements have been done. No software should avoid using character instructions in anticipation of them being subsetted in a future low end machine; otherwise, the performance data will be biased.

E.5 GUIDELINES TO HARDWARE IMPLEMENTORS

E.5.1 First VAX-11 Machine

All instructions in the architecture will be provided and no instructions will be subsettable or provided as processor options. G_floating and H_floating instructions were added to the architecture after the first machine was designed and shipped; therefore they will be offered as options.

E.5.2 Machines After The First

Any or all instructions in each omitted instruction class (classes 1-8) may be omitted or provided as an option. No kernel instructions may be omitted or included as an option. If any subsettable instructions are omitted, the processor implementors are required to provide simulation routines for omitted instructions. Then quantifiable space, time, and cost tradeoffs can be made by hardware and software implementors early in the processor design cycle by measuring existing systems.

E.5.3 Later Additions To The Architecture

Additions to the architecture after April, 1976, will not necessarily be in all machines. Thus, the specification of the instruction must include the simulation code for older machines.

[End of Appendix E]

Title: VAX-11 Instruction Set and Opcode Assignments -- Rev 16

Specification Status: Fully approved

Architectural Status: under ECO control

File: SRFR16.RNO

PDM #: not used

Date: 25-Oct-78

Superseded Specs: Rev 15

Author: P. Conklin & A. Helenius

Typist: L. Principe

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Appendix F describes the format of all of the instructions including the data type of each operand. It also includes the opcode assignment of each instruction. This document is updated in parallel with every ECO to the VAX-11 Architecture.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Review of Chapters 1-4 SRM Rev 1 ECO #1	Hastings	19-Dec-75
Rev 2	Results of Pruning ECO #2	Hastings	7-Jan-76
Rev 3	Chapter 10 ECO #3	Rothman	9-Jan-76
Rev 4	Changes to Field, Loop, and String ECO #4	Rodgers	28-Feb-77
Rev 5	Name Changes ECO #5	Strecker	2-Feb-76
Rev 6	Memory Management ECO #6	Hastings	
Rev 7	Opcode Assignments Added	Rodgers	22-Jan-76
Rev 8	(ECO #8 - Chapter 6 does not change this)		
Rev 9	Conditional Branch ECO #9	Strecker	13-Feb-76
Rev 10	Change to Make Same as SRM	Strecker	26-Feb-76
Rev 11	Proofreading Rev 10	Strecker	27-Feb-76
Rev 12	CALL ECO #11	Gourd	25-Feb-76
Rev 13	Field Instructions ECO #10	Strecker	16-Mar-76
Rev 14	Per April 1-9 Meeting	Strecker	3-Jun-76
Rev 15	ECO's	Strecker	9-Mar-77
Rev 16	Interlocked queue, Octaword, G_floating, and H_floating	Bhandarkar	25-Oct-78

Rev 15 to Rev 16:

1. Add 2-byte opcodes for G_floating and H_floating.
2. Fix CRC destination.
3. Fix POLYD implied operands.
4. Add interlocked queue instructions.

Rev 14 to Rev 15:

1. Change name of EDITN to EDITPC (EDITPC ECO).
2. Delete option and destination length operands of EDIT (EDITPC ECO).
3. Change from zoned to packed instructions (decimal data ECO).
4. Add CLRF and CLRD as same as CLRL and CLRQ.
5. Add Rounding to ASHP.
6. Correct order of field reference on EXTV, etc.
7. Change operands of POLY Instruction
8. Add CVT{SP,PS}; change xxN to xxT; add INDEX.
9. Add ADAWI.

Rev 13 to Rev 14:

1. Add MOVQ
2. Remove CLRF
3. Add ADDA2, ADDA3, SUBA2, SUBA3
4. Remove CHOPF, CHOPD
5. Correct EMOD operands per ECO 18
6. Remove POLYF, POLYD
7. Change MOVP, PUSHP to MOVA, PUSHA
8. Remove CMPA, DIFA, ADTA, SBFA
9. Correct conditional branches per ECO 17
10. Add BBSSI, BBCCI

11. Remove MSx, MSPx

12. Add XFC

13. Remove ECMA

14. Add INSQUE, REMQUE

15. Add MOVTUC, MATCHC

16. Add CRC per ECO 12

17. Correct CRC operands

18. Remove MOVU, CVTLU, CVTPU, ASHU; change names to MOVN, CVTLN, CVTPN, ASHN

19. Remove MULN4, DIVN4; change names to MULN, DIVN

20. Change name to EDITN

21. Add CHMK, CHME, CHMS, CHMU, CHMI

22. Add PRBPR, PRBPRW

23. Add LDPC, SVPC

24. Add MTPR, MFPR

25. Replace opcode encoding to minimize decode logic (Apr 76)

26. Change names to LDPCTX, SVPCTX

27. Change MT/FPR from .w/rl to .rl

28. Change queue context from .aq to .ab

29. Correct FFS/C to .wl

30. Correct EMOD to mulrx

31. Correct typo on CLR

32. Add section on instruction interruptiblility

33. Change operand names to agree with chapter 4-9

34. Add mode to PROBER/W and change names

35. Remove CHMI

36. Change MOVPSW to MOVPSL

37. Remove ADDA, SUBA

38. HALT and BPT are faults

39. Add POLYF/D

40. Add SKPC

41. Change EMODD to int.wl

42. Add implicit operands

43. Change field operand type from .ab to .vb

44. Change name to BLBS/C

45. BPT is legal in all modes and on interrupt stack

46. SVPCTX is legal in kernel mode

47. Add final encoding (1-Jun-76)

48. Add CRC destination

49. Expand I/O restriction rules

50. Allow boot in I/O space

51. Put {} around implied operands

52. Add field implied operand on field instructions

53. Add register implied operands on string and POLY

Rev 12 to Rev 13:

1. Change operands to EXTV, EXTZV, INSV, CMPV, CMPZV per ECO 10

2. Change operands of FFC, FFS per ECO 10

3. Change operands of BBx, BBxx per ECO 10

4. Change operands to BLS, BLC per ECO 10

Rev 11 to Rev 12:

1. Change operand of CALLS per ECO 11

Rev 7 to Rev 11:

1. Add BME, BPN

2. Remove BNEVER

3. Add BSBB; change name to BSBW

4. Change name to BRB, BRW

5. Change opcode assignments of above and BLS, BLC, JMP, JSB

Rev 5 to Rev 7:

1. Add opcode assignments; add CRLF

Rev 4 to Rev 5:

1. Change name to EMUL, EDIV

2. Change name to ASHQ

3. Change name to MOVP, PUSHP, CMPA, ADTA, SBFA, DIFA

4. Change name to FFC, FFS

5. Change name to BR1, BR2

6. Change name to BLS, BLC

7. Change name to CALLG, RET

8. Change name to MOVPSW

9. Change name to EMOD

10. Change name to MOVC, MOVTC, CMPC, SCANC, SPAN, LOCC

11. Change name to CMPN, ADDN, SUBN, MULN, DIVN, CVTNL, CVTLS, CVTPS, ASHS

12. Change name to MOVC3, MOVC5, etc.

13. Change name to EDITC

14. Change name to MS, MPS

Rev 1 to Rev 4:

1. Change operands of INSV, EXTV, EXTZV, CMPV, CMPZV, FFZ, FFO, BBx, BBxx, BT, BF

2. Remove AVP

3. Change order of string descriptor

4. Change order of operands in ACB, AOBLT, SOBGT

[End of SRFR16.RNO]

APPENDIX F

INSTRUCTION SET AND OPCODE ASSIGNMENTS

25-Oct-78 -- Rev 16

F.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. See Appendix B, VAX-11 assembler notation, for a description of when the data type suffix and operand number suffix may be omitted. The order of the instructions is generally that in the SRM.

		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	40

8.	Convert Rounded CVTR{F,D,G,H}L src.rx, dst.wl	4
9.	Compare CMP{B,W,L,F,D,G,H} src1.rx, src2.rx	7
10.	Test TST{B,W,L,F,D,G,H} src.rx	7
11.	Add 2 Operand ADD{B,W,L,F,D,G,H}2 add.rx, sum.mx	7
12.	Add 3 Operand ADD{B,W,L,F,D,G,H}3 add1.rx, add2.rx, sum.wx	7
13.	Increment INC{B,W,L} sum.mx	3
14.	Add With Carry ADWC add.rl, sum.ml	1
15.	Add Aligned Word ADAWI add.rw, sum.mw	1
16.	Subtract 2 Operand SUB{B,W,L,F,D,G,H}2 sub.rx, dif.mx	7
17.	Subtract 3 Operand SUB{B,W,L,F,D,G,H}3 sub.rx, min.rx, dif.wx	7
18.	Decrement DEC{B,W,L} dif.mx	3
19.	Subtract With Carry SBWC sub.rl, dif.ml	1
20.	Multiply 2 Operand MUL{B,W,L,F,D,G,H}2 mulr.rx, prod.mx	7
21.	Multiply 3 Operand MUL{B,W,L,F,D,G,H}3 mulr.rx, muld.rx, prod.wx	7
22.	Extended Multiply EMUL mulr.rl, muld.rl, add.rl, prod.wq	1
23.	Divide 2 Operand DIV{B,W,L,F,D,G,H}2 divr.rx, quo.mx	7
24.	Divide 3 Operand DIV{B,W,L,F,D,G,H}3 divr.rx, divd.rx, quo.wx	7
25.	Extended Divide EDIV divr.rl, divd.rq, quo.wl, rem.wl	1

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

30. 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, {R0-3.wl}

37. Polynomial Evaluation D_floating 1
POLYD arg.rd, degree.rw, tbladdr.ab, {R0-5.wl}

38. Polynomial Evaluation G_floating 1
POLYG arg.rg, degree.rw, tbladdr.ab, {R0-5.wl}

39. Polynomial Evaluation H_floating 1
POLYH arg.rh, degree.rw, tbladdr.ab,
{R0-5.wl,-16(SP):-1(SP).wb}

40. 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, ~~G,H~~} 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, G, H} 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

80. 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, {R0-5.wl}

84. Move Character 5 operand 1
MOVC5 srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab,
{R0-5.wl}

85. Move Translated Characters 1
MOVTC srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw,
dstaddr.ab, {R0-5.wl}

86. Move Translated Until Character 1
MOVTUC srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen,rw,
dstaddr.ab, {R0-5.wl}

87. Compare Characters 3 Operand 1
CMPC3 len.rw, src1addr.ab, src2addr.ab, {R0-3.wl}

88. Compare Characters 5 Operand 1
CMPC5 src1len.rw, src1addr.ab, fill.rb, src2len.rw, src2addr.ab, {R0-3.wl}

89. Scan Characters 1
SCANC len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.wl}

90. Span Characters 1
SPANC len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.wl}

91. Locate Character 1
LOCC char.rb, len.rw, addr.ab, {R0-1.wl}

92. Skip Character 1
SKPC char.rb, len.rw, addr.ab, {R0-1.wl}

93. Match Characters 1
MATCHC len1.rw, addr1.ab, len2.rw, addr2.ab, {R0-3.wl}

94. Cyclic Redundancy Check 1
CRC tbl.ab, inicrc.rl, strlen.rw, stream.ab, {R0-3.wl}

95. Move Packed 1
MOVP len.rw, srcaddr.ab, dstaddr.ab, {R0-3.wl}

96. Compare Packed 3 Operand 1
CMPP3 len.rw, src1addr.ab, src2addr.ab, {R0-3.wl}

97. Compare Packed 4 Operand 1
CMPP4 src1len.rw, src1addr.ab, src2len.rw, src2addr.ab, {R0-3.wl}

98. Add Packed 4 Operand 1
ADDP4 addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab, {R0-3.wl}

99. Add Packed 6 Operand 1
ADDP6 add1len.rw, add1addr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab, {R0-5.wl}

100. Subtract Packed 4 Operand 1
SUBP4 sublen.rw, subaddr.ab, diflen.rw, difaddr.ab, {R0-3.wl}

101. Subtract Packed 6 Operand 1
SUBP6 sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, diflen.rw, difaddr.ab, {R0-5.wl}

102. Multiply Packed 1
MULP mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab, {R0-5.wl}

103. Divide Packed 1
DIVP divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quolen.rw, quoaddr.ab, {R0-5.wl, -16(SP):-1(SP).wb}

104. Convert Long to Packed 1
CVTLP src.rl, dstlen.rw, dstaddr.ab, {R0-3.wl}

105. Convert Packed to Long 1
CVTPL srclen.rw, srcaddr.ab, {R0-3.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, {R0-3.wl}

107. Convert Packed to Leading Separate 2
Convert Leading Separate to Packed
CVT{PS,SP} srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab, {R0-3.wl}

108. Arithmetic Shift and Round Packed 1
ASHP cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab, {R0-3.wl}

109. Edit Packed to Character String 1
EDITPC srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab, {R0-5.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 1
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 304

F.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

F.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

OPCODE ASSIGNMENTS

Binary	Hex	Mnemonic	Binary	Hex	Mnemonic
01000000	40	ADDF2	01100000	60	ADDD2
01000001	41	ADDF3	01100001	61	ADDD3
01000010	42	SUBF2	01100010	62	SUBD2
01000011	43	SUBF3	01100011	63	SUBD3
01000100	44	MULF2	01100100	64	MULD2
01000101	45	MULF3	01100101	65	MULD3
01000110	46	DIVF2	01100110	66	DIVD2
01000111	47	DIVF3	01100111	67	DIVD3
01001000	48	CVTFB	01101000	68	CVTDB
01001001	49	CVTFW	01101001	69	CVTDW
01001010	4A	CVTFL	01101010	6A	CVTDL
01001011	4B	CVTRFL	01101011	6B	CVTRDL
01001100	4C	CVTBF	01101100	6C	CVTBD
01001101	4D	CVTWF	01101101	6D	CVTWD
01001110	4E	CVTLF	01101110	6E	CVTLD
01001111	4F	ACBF	01101111	6F	ACBD
01010000	50	MOVF	01110000	70	MOVD
01010001	51	CMPF	01110001	71	CMPD
01010010	52	MNEGF	01110010	72	MNEGD
01010011	53	TSTF	01110011	73	TSTD
01010100	54	EMODF	01110100	74	EMODD
01010101	55	POLYF	01110101	75	POLYD
01010110	56	CVTFD	01110110	76	CVTDF
01010111	57	RESERVED to DEC	01110111	77	RESERVED to DEC
01011000	58	ADAWI	01111000	78	ASHL
01011001	59	RESERVED to DEC	01111001	79	ASHQ
01011010	5A	RESERVED to DEC	01111010	7A	EMUL
01011011	5B	RESERVED to DEC	01111011	7B	EDIV
01011100	5C	INSQHI	01111100	7C	CLRQ,CLRD,CLRG
01011101	5D	INSQTI	01111101	7D	MOVQ
01011110	5E	REMQHI	01111110	7E	MOVAQ,MOVAD,MOVAG
01011111	5F	REMQTI	01111111	7F	PUSHAQ,PUSHAD,PUSHAG

OPCODE ASSIGNMENTS

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	Binary	Hex	Mnemonic
11000000	C0	ADDL2	11100000	E0	BBS
11000001	C1	ADDL3	11100001	E1	BBC
11000010	C2	SUBL2	11100010	E2	BBSS
11000011	C3	SUBL3	11100011	E3	BBCS
11000100	C4	MULL2	11100100	E4	BBSC
11000101	C5	MULL3	11100101	E5	BBCC
11000110	C6	DIVL2	11100110	E6	BBSSI
11000111	C7	DIVL3	11100111	E7	BBCCI
11001000	C8	BISL2	11101000	E8	BLBS
11001001	C9	BISL3	11101001	E9	BLBC
11001010	CA	BICL2	11101010	EA	FFS
11001011	CB	BICL3	11101011	EB	FFC
11001100	CC	XORL2	11101100	EC	CMPV
11001101	CD	XORL3	11101101	ED	CMPZV
11001110	CE	MNEGL	11101110	EE	EXTV
11001111	CF	CASEL	11101111	EF	EXTZV
11010000	D0	MOVL	11110000	F0	INSV
11010001	D1	CMPL	11110001	F1	ACBL
11010010	D2	MCOML	11110010	F2	AOBLSS
11010011	D3	BITL	11110011	F3	AOBLEQ
11010100	D4	CLRL,CLRF	11110100	F4	SOBGEQ
11010101	D5	TSTL	11110101	F5	SOBGTR
11010110	D6	INCL	11110110	F6	CVTLB
11010111	D7	DECL	11110111	F7	CVTLW
11011000	D8	ADWC	11111000	F8	ASHP
11011001	D9	SBWC	11111001	F9	CVTLP
11011010	DA	MTPR	11111010	FA	CALLG
11011011	DB	MFPR	11111011	FB	CALLS
11011100	DC	MOVPSL	11111100	FC	XFC
11011101	DD	PUSHL	11111101	FD	ESCD to DEC
11011110	DE	MOVAL,MOVAF	11111110	FE	ESCE to DEC
11011111	DF	PUSHAL,PUSHAF	11111111	FF	ESCF to DEC

TWO BYTE OPCODES

Hex	Mnemonic	Hex	Mnemonic
00FD to 31FD	RESERVED to DIGITAL		
32FD	CVTDH	33FD	CVTGF
34FD to 3FFD	RESERVED to DEC		
40FD	ADDG2	60FD	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

50FD	MOVG	70FD	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
80FD 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		

F.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

In any case, the programmer is responsible for ensuring that any memory reference to I/O space is in an instruction which can not take an exception after the first I/O space reference. This includes deferred references to I/O space.

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)
```

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.

\ For reference, instructions were discarded as follows:

1. String: MOVC3/5, MOVTC, MOVTUC, CMPC3/5, SCANC, SPANC, LOCC, SKPC, MATCHC, CRC, MOVP, CMPP3/4, ADDP4/6, SUBP4/6, MULP, DIVP, CVTLP, CVTPL, CVTPT, CVTTP, CVTPS, CVTSP, ASHP, EDITPC

2. not in kernel set: MOV{F,D,G,H}, MNEG{F,D,G,H}, CVT{B,W,L,F,D,G,H}{F,D,G,H}, CVT{F,D,G,H}{B,W,L,F,D,G,H}, CVTR{F,D,G,H}L, CMP{F,D,G,H}, TST{F,D,G,H}, ADD{F,D,G,H}2, ADD{F,D,G,H}3, SUB{F,D,G,H}2, SUB{F,D,G,H}3, MUL{F,D,G,H}2, MUL{F,D,G,H}3, DIV{F,D,G,H}2, DIV{F,D,G,H}3, EMOD{F,D,G,H}, POLY{F,D,G,H}, ACB{F,D,G,H}

3. PC, SP, PCBB not in I/O space and instruction has no other operands: Bxxx, BRB/W, JMP, BSBB/W, JSB, RSB, RET, BPT, REI, HALT, NOP, LDPCTX, SVPCTX

4. operand types: MOVQ, CLRQ, ASHQ, INSQUE, BB{S,C}, BB{S,C}{S,C}, BB{SS,CC}I

5. slow: MUL{B,W,L}2, MUL{B,W,L}3, EMUL, DIV{B,W,L}2, DIV{B,W,L}3, EDIV, ASHL, ROTL, EXTV, EXTZV, INDEX, INSV, CMPV, CMPZV, FFS/C, CALLG/S, PUSHR, POPR

6. modify or write operand must be last: ACB{B,W,L}, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR

\

[End of Appendix F]

APPENDIX G

Unwritten

Title: VAX-11 Multiprecision Arithmetic -- Rev 4

Specification Status: Fully approved

Architectural Status: under ECO control

File: SRHR4.RNO

PDM #: not used

Date: 21-Mar-77

Superseded Specs:

Author: P. Conklin

Typist: B. Call

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Appendix H shows how to use the instruction set to perform general multiprecision integer arithmetic. In practice, these algorithms would be tuned to handle specific environments such as variable precision.

Revision History:

Rev.#	Description	Author	Revised Date
Rev 3	Original	Conklin	12-May-76
Rev 4	Typos	Conklin	21-Mar-77

Rev 3 to Rev 4:

1. Typos.

2. Correct order of looping operands.

Rev 1 to Rev 3:

1. Orignal creation

[END OF APPENDIX H]

[End of SRHR4.RNO]

APPENDIX H

MULTIPRECISION ARITHMETIC

21-Mar-77 -- Rev 4

H.1 OVERVIEW

Multiprecision integers are stored as a sequence of consecutive bytes in memory. They can be thought of as a set of N consecutive longwords. The sequence is a two's complement representation with bit 31 of the highest longword as the sign. Arithmetic is performed without integer overflow trapping using the ADWC, SBWC, EMUL, and EDIV instructions. Low order components are treated as unsigned positive integers with the C condition code containing the carry. If the V condition code is set after computing the high order result, then the result overflowed. No attempt is made to set the Z condition code correctly.

For the purpose of this section, the data type is "M" for multiprecision. Each of the integer arithmetic instructions is composed as a pattern. In all cases, N is the common length of the operands in longwords.

H.2 ADDM2 ADD,SUM

```
        BICPSW  #^X21           ;clear carry and integer overflow
        CLRL    R0              ;clear loop index
 1$:    ADWC    ADD[R0],SUM[R0] ;add including carry
        AOBLSS  #N,R0,,1$       ;loop over vector
        BVS     overflow        ;branch if overflow
```

H.3 ADDM3 ADD,AUG,SUM

```
        BICPSW  #^X21           ;clear carry and integer overflow
        CLRL    R0              ;clear loop index
1$:     MOVL    ADD[R0],SUM[R0] ;move addend to result
        ADWC    AUG[R0],SUM[R0] ;add including carry
        AOBLSS  #N,R0,1$        ;loop over vector
        BVS     overflow        ;branch if overflow
```

H.4 SUBM2 SUB,DIF

```
        BICPSW  #^X21           ;clear carry and integer overflow
        CLRL    R0              ;clear loop index
1$:     SBWC    SUB[R0],DIF[R0] ;subtract including borrow
        AOBLSS  #N,R0,1$        ;loop over vector
        BVS     overflow        ;branch if overflow
```

H.5 SUBM3 SUB,MIN,DIF

```
        BICPSW  #^X21           ;clear carry and integer overflow
        CLRL    R0              ;clear loop index
1$:     MOVL    SUB[R0],DIF[R0] ;move subtrahend to result
        SBWC    MIN[R0],DIF[R0] ;subtract including borrow
        AOBLSS  #N,R0,1$        ;loop over vector
        BVS     overflow        ;branch if overflow
```

H.6 EMULM MULR,MULD,PROD

```
        BICPSW  #^X20           ;clear integer overflow
        CLRL    R4              ;clear destination loop index
        CLRQ    R2              ;clear temporary accumulator
1$:     MOVL    R2,PROD[R4]     ;start result with carry sum
        MOVL    R3,R2           ;adjust carry
        CLRL    R3              ;clear excess carry
        MOVL    R4,R6           ;set inner down count
        COMPL   R6,#N-1         ;see if beyond end
        BLEQ    2$              ;no--ok to proceed
        MOVL    #N-1,R6         ;yes--back up to end
2$:     SUBL3   R4,R6,R5        ;clear inner up count
3$:     EMUL    MULR[R5],MULD[R6],#0,R0 ;extended multiply
        TSTL    MULR[R5]        ;handle unsigned MULR longword
        BGEQ    4$              ;same as signed--no fixup
        ADDL2   MULD[R6],R1     ;different--fix result
4$:     TSTL    MULD[R6]        ;handle unsigned MULD longword
        BGEQ    5$              ;same as signed--no fixup
        ADDL2   MULR[R5],R1     ;different--fix result
5$:     ADDL2   R0,PROD[R4]     ;incorporate product
        ADWC    R1,R2           ;count extension as carry
        ADWC    #0,R3           ;(quad carry)
        AOBLEQ  #N-1,R5,6$      ;advance count
        BRB     7$              ;terminate loop if done
6$:     SOBGEQ  R6,3$           ;loop for this result
7$:     AOBLEQ  #2*N-2,R4,1$    ;loop over result vector
        MOVL    R2,PROD+2*N-1   ;store final result
        TSTL    MULR+N-1        ;handle signed MULR vector
        BGEQ    9$              ;same as signed--no fixup
        CLRL    R4              ;different--clear loop index
8$:     SBWC    MULD[R4],PROD+N[R4] ;subtract to fix result
        AOBLEQ  #N-1,R4,8$      ;loop over fixup
9$:     TSTL    MULD+N-1        ;handle signed MULD vector
        BGEQ    11$             ;same as signed--no fixup
        BICPSW  #1              ;clear carry
        CLRL    R4              ;different--clear loop index
10$:    SBWC    MULR[R4],PROD+N[R4] ;subtract to fix result
        AOBLEQ  #N-1,R4,10$     ;loop over fixup
11$:    TSTL    PROD+2N-1       ;set condition code N
```

APPENDIX I

PDP-11 TO VAX-11 CONVERSION GUIDE

24-Mar-77-- Rev 4

The following is an aid to converting PDP-11 programs to VAX-11 programs. Each PDP-11 instruction is paired with its VAX-11 equivalent. Five types of equivalence are noted:

1. Equivalent instruction.

2. Possible simulation. A sequence of VAX-11 instructions is equivalent. There is no suggestion that the simulation given is the best in space or time. The stack is used for temporary space in the simulations. Particular care must be taken with respect to operands specified by addressing modes with side effects.

3. Functionality. A VAX instruction provides similar functionality. The programmer should compare the VAX-11 and PDP-11 instruction.

4. No good simulation. Although the instruction can be simulated, the space or time required is disproportionate to the probable contribution of the instruction to the algorithm being implemented. The programmer will generally find another way to implement the algorithm.

5. Not available. Instruction relates to concepts not in VAX-11 architecture.

In addition, note that there is no odd address trap.

PDP-11	VAX-11
HALT	HALT. Saved PC different; canonical PSL set on fault.
WAIT	Not available.
RTI	Functionality available with REI. Faults if PSL inward.
BPT	BPT. Saved PC different; canonical PSL set.
IOT	Functionality available with CHMK, CHME, CHMS, CHMU. Canonical PSL set.
RESET	Functionality available with MTPR. Faults outside kernel mode.
RTT	Functionality available with REI.
JMP	JMP.
RTS Rn	If n EQL 7, RSB. Otherwise no good simulation. However, certain sequences including RTS may be simulated very efficiently using RET.
SPL	Functionality available with MTPR. Faults outside kernel mode.
NOP	NOP.
CLear CC	BICPSW.
SEt CC	BISPSW.

SWAB dst	Possible simulation:
	
	MOVW dst,-(SP)
	MOVB (SP),1(SP)
	MOVB dst,(SP)
	MOVW (SP)+,dst
	
	N,Z,V,C affected differently.

BR	BRB or BRW.
BNE	BNEQ.

BEQ — BEQL.

BGE — BGEQ. Branches on N EQL 0 rather than N XOR V EQL 0.

BLT — BLSS. Branches on N EQL 1 rather than N XOR V EQL 1.

BGT — BGTR. Branches on Z OR N EQL 0 rather than Z OR {N XOR V} EQL 0.

BLE — BLEQ. Branches on Z OR N EQL 1 rather than Z OR {N XOR V} EQL 1.

JSR Rn,dst — If n=7, BSBB, BSBW, or JSB.

Otherwise no good simulation. However certain sequences including JSR may be simulated very efficiently using CALLS or CALLG.

CLR — CLRW. C affected differently.

COM dst — MCOMW dst,dst. C affected differently.

INC — INCW. C affected differently.

DEC — DECW. C affected differently.

NEG dst — MNEGW dst,dst.

ADC dst — Possible simulation:

```
        BCC     1$
        INCW    dst
    1$:
```

V, C affected differently.

In most cases ADC appears in sequences like:

```
        ADD     A,B
        ADC     B+2
        ADD     A+2,B+2
```

This is of course simulated by:

```
        ADDL    A,B
```

SBC dst

Possible simulation:

```
        BCC     1$
        DECW    dst
    1$:
```

V, C affected differently.

In most cases SBC appears in sequences like:

```
        SUB     A,B
        SBC     B+2
        SUB     A+2,B+2
```

This is of course simulated by:

```
        SUBL    A,B
```

TST

TSTW.

ROR

No good simulation.

ROL

No good simulation.

ASR dst

Possible simulation:

```
        CVTWL   dst,-(SP)
        ASHL    #-1,(SP),(SP)
        CVTLW   dst
```

V, C affected differently.

ASL dst

ADDW2 dst,dst.

MARK

Not available. Stack clean up functionality provided by CALLS/RET.

MFPI src

Not available. Functionality available with MOVW src,-(SP).

MTPI dst

Not available. Functionality available with MOVW (SP)+,dst.

SXT dst

Possible simulation:

```
        BLSS    1$
        CLRW    dst
        BRB     2$
    1$: MNEGW   #1,dst
    2$:
```

C affected differently.

MOV — MOVW.

CMP — CMPW. N, V affected differently. However compare followed by an equivalent conditional branch behaves similarly.

BIT — BITW.

BIC — BICW2.

BIS — BISW2.

ADD — ADDW2.

MUL src,Rn — If n is odd:

```
MULW2   src,Rn
```

V, C affected differently.

If n is even, possible simulation:

```
CVTWL   Rn,-(SP)
CVTWL   src,-(SP)
MULL2   (SP)+,(SP)
MOVW    (SP)+,Rn+1
MOVW    (SP)+,Rn
```

V, C affected differently.

If the functionality of high and low halves of result in different registers is not needed, possible simulation:

```
CVTWL   Rn,Rn
CVTWL   src,-(SP)
MULL2   (SP)+,Rn
```

V, C affected differently.

DIV src,Rn — Possible simulation:

```
MOVW    Rn,-(SP)
MOVw    Rn!1,-(SP)
CVTWL   src,-(SP)
EDIV    (SP)+,(SP)+,Rn,Rn!1
```

V, C affected differently.

If only the functionality of the quotient is needed:

```
DIVW2   src,Rn
```

V, C affected differently.

ASH src,Rn — Possible simulation:

```
CVTWL   Rn,Rn
ASHL    src,Rn,Rn
```

V, C affected differently. Result may be different if src GTR 31 or src LSS -31.

ASHC src,Rn — If n even, possible simulation:

```
ROTL    #16,Rn,Rn
MOVW    Rn+1,Rn
ASHL    src,Rn
MOVW    Rn,Rn+1
ROTL    #16,Rn,Rn
```

V, C affected differently.

If the functionality of high and low halves in different registers is not needed:

```
ASHL    src,Rn,Rn
```

Result may be different if src GTR 31 or src LSS -31.

If n odd:

```
INSV    Rn,#16,#16,Rn
ASHL    src,Rn
```

V, C affected differently.

XOR	XORW2.
FADD Rn	ADDF2 (Rn)+,(Rn). Condition codes different on overflow.
FSUB Rn	SUBF2 (Rn)+,(Rn). Condition codes different on overflow.
FMUL Rn	MULF2 (Rn)+,(Rn). Condition codes different on overflow.
FDIV Rn	DIVF2 (Rn)+,(Rn). Condition codes different on overflow or divide by zero.
SOB Rn,dst	Possible simulation: DECW Rn BNE dst N, Z, V affected differently. Similar functionality available with SOBGTR.
BPL	BGTR.
BMI	BLSS.
BHI	BGTRU.
BLOS	BLEQU.
BVC	BVC.
BVS	BVS.
BCC, BHIS	BCC, BGEQU.
BCS, BLO	BCS, BLSSU.
EMT	Functionality available with CHMK, CHME, CHMS, CHMU. Canonical PSL set.
TRAP	Functionality available with CHHK, CMME, CHMS, CHMU. Canonical PSL set.
CLRB	CLRB. C affected differently.
COMB dst	MCOMB dst,dst. C affected differently.
INCB	INCB. C affected differently.

DECB — DECB. C affected differently.

NEGB dst — MNEGB dst,dst.

ADCB dst — Possible simulation:

```
        BCC     1$
        INCB    dst
    1$:
```

V, C affected differently.

SBCB dst — Possible simulation:

```
        BCC     1$
        DECB    dst
    1$:
```

V, C affected differently.

TSTB — TSTB.

RORB — No good simulation.

ROLB — No good simulation.

ASRB dst — Possible simulation:

```
        CVTBL   dst,-(SP)
        ASHL    #1,(SP),(SP)
        CVTLB   (SP)+,dst
```

V, C affected differently.

ASLB dst — ADDB2 dst,dst.

MTPS src — Usually just BISPSW, BICPSW.
Possible simulation:

```
        MOVPSL  -(SP)
        MOVB    src,(SP)
        MOVW    (SP)+,(SP)
        BICPSW  #-1
        BISPSW  (SP)+
```

T affected differently.

MFPD src	Not available. Functionality available with MOVW src,-(SP).
MTPD dst	Not available. Functionality available with MOVW (SP)+,dst.
MFPS dst	Possible simulation: MOVPSL -(SP) INSV #0,#8,(SP)+,dst dst<7:5> different. N, Z affected differently
MOVB src,dst	If dst is not a register, MOVB. If dst is a register, CVTBW.
CMPB	CMPB. N, V affected differently. However compare followed by an equivalent conditional branch behaves similarly.
BITB	BITB.
BICB	BICB2.
BISB	BISB2.
SUB	SUBW2.
CFCC	Not available. Only a single set of condition codes.
SETF	Not available. Separate instructions.
SETI	Not available. Separate instructions.
SETD	Not available. Separate instructions.
SETL	Not available. Separate instructions.
LDFPS	Not available. Only one status word.
STFPS	Not available. Only one status word.
STST	Not available. Only one status word.
CLRF	CLRF. No FCC.
CLRD	CLRD. No FCC.
TSTF	TSTF. No FCC.
TSTD	TSTD. No FCC.

ABSF src,dst Possible simulation:

```
        MOVF    src,dst
        BGEQ    1$
        MNEGF   dst,dst
    1$:
```

No FCC.

ABSD src,dst Possible simulation:

```
        MOVD    src,dst
        BGEQ    1$
        MNEGD   dst,dst
    1$:
```

No FCC.

NEGF dst MNEGF dst,dst. No FCC.

NEGD dst MNEGD dst,dst. No FCC.

MULF MULF2. No FCC.

MULD MULD2. No FCC.

MODF Not available. Functionality provided by EMODF.

MODD Not available. Functionality provided by EMODD.

ADDF ADDF2. No FCC.

ADDD ADDD2. No FCC.

LDF MOVF. No FCC.

LDD MOVD. No FCC.

SUBF SUBF2. No FCC.

SUBD SUBD2. No FCC.

CMPF CMPF. No FCC.

CMPD CMPD. No FCC.

STF MOVF. No FCC.

STD	MOVD. No FCC.
DIVF	DIVF2. No FCC.
DIVD	DIVD2. No FCC.
STEXP Rn,dst	Possible simulation: EXTV #7,#8,Rn,-(SP) CVTLB (SP)+,-(SP) SUBB3 #128,(SP)+,dst V, C affected differently. No FCC.
STCFI	CVTFW. C, V affected differently. No FCC.
STCFL	CVTFL. C, V affected differently. No FCC. Longword format different.
STCDI	CVTDW. C, V affected differently. No FCC.
STCDL	CVTDL. C, V affected differently. No FCC. Longword format different.
STCFD	CVTFD. No FCC.
STCDF	CVTDF. No FCC.
LDEXP src,Rn	Possible simulation: SUBL2 #3,SP ADDB3 #128,src,-(SP) INSV (SP)+,#7,#8,Rn No FCC.
LDCIF	CVTWF. No FCC.
LDCID	CVTWD. No FCC.
LDCLF	CVTLF. No FCC. Longword format different.
LDCLD	CVTLD. No FCC. Longword format different.
LDCDF	CVTDF. No FCC.
LDCFD	CVTFD. No FCC.

[End of Appendix I]

Title: PDP-11 to VAX-11 Conversion Guide -- Rev 4

Specification Status: Fully approved

Architectural Status: under ECO control

FILE: SRIR4.RNO

PDM #: not used

Date: 24-Mar-77

Superseded Specs:

Author: W. Strecker & P. Conklin

Typist: B. Call

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: Appendix I is an aid to converting PDP-11 programs to VAX-11 programs. Each PDP-11 instruction is paired with its VAX-11 equivalent. The types of equivalence are:

1. Equivalent instruction
2. Possible simulation given
3. Functional similarity given
4. No good simulation
5. Not available

The table is in the order of the PDP-11 instruction opcode assignments.

Revision History:

Rev.#	Description	Author	Revised Date
Rev 1	Notes	Karich	Oct-75
Rev 2	Corrected Notes	Conklin	Jan-76
Rev 3	Formalized, and Reconciled	Strecker	13-Jun-76
Rev 4	MTPS, MFPS	Strecker	24-Mar-77

Rev 3 to Rev 4:

1. Add MTPS, MFPS.

Rev 2 to Rev 3:

1. Formalize the various categories
2. Add complete instruction list
3. Include many possible simulations
4. Reconcile with SRM Rev 3
5. Improve simulation of ASL, ASLB

Rev 1 to Rev 2:

1. Correct various misunderstandings
2. Reconcile with SRM Rev 2

[End of SRIR4.RNO]

Title: VAX-11 Address Validation Rules - Rev 4

Specification Status: Fully Approved

Architectural Status: under ECO control

File: SRJR4.RNO

PDM #: not used

Date: 1-Feb-77

Superseded Specs:

Author: D. Cutler

Typist: J. Bess

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: The hardware memory management mechanisms described in Chapter 5 must be supplimented with software to provide a protected operating environment. Appendix J sets forth hardware and software assumptions about such an environment and a set of rules that must be followed by operating system software to construct a protected system.

Revision History:

Rev.#	Description	Author	Revised Date
Rev 1	Original	Cutler	20-May-76
Rev 2	Incorporate Review Comments	Cutler	8-Jun-76
Rev 3	skipped to maintain numbering		
Rev 4	Typos	Cutler	1-Feb-77

Rev 2 to Rev 4:

1. Typos.

2. Require PROBE before read to protect I/O side effects.

Rev 1 to Rev 2:

1. Add retriction on MTPR to previous mode of PSL.

2. Add assumption that REI verifies consistency.

[End of SRJR4.RNO]

APPENDIX J

ADDRESS VALIDATION RULES

1-Feb-77 -- Rev 4

The memory management system described in Chapter 5 separates validation from the access of arguments. In the previous scheme validation and access were performed as an indivisible operation. There is some question as to whether the new scheme will be adequate to build reliable and secure operating systems. Specifically will it be possible for a user to call an inner access mode in such a manner as to cause the inner access mode to access data in a way that corrupts system integrity (e.g., causes supervisory code to write over itself) or incorrectly allows access to data that would otherwise have been inaccessible (e.g., the reading of a password table).

In order to accomplish either of these encroachments, the user must be able to generate a bogus address in such a way to cause supervisory software to have protection or security holes. The following discussion sets forth operating system and hardware assumptions and then explains the rules that must be adhered to when accessing arguments from an inner access mode to avoid such a hole.

The following assumptions are made concerning operating system software:

1. Operating system software (kernel and executive mode) is trustworthy and does not maliciously attempt to breakdown the protection mechanisms (e.g., change the mapping or protection of pages at arbitrary times).

2. The protection of a shared page may not be changed unless the share count is one and the process attempting the change is that sharer.

3. The protection of a page with a nonzero I/O pending count may not be changed until the count goes to zero.

4. Operating system software will not deliver AST's to outer access modes while the process is executing in an inner access mode.

5. Arguments passed to an inner access mode can be maliciously destroyed asynchronously by another process (e.g., shared data) or by an I/O transfer, but not by a less privileged mode of the executing process itself.

6. Kernel and executive stacks are never allocated in shared memory or accessible to other than their respective access modes.

The following assumptions are made concerning the VAX hardware:

1. Four access modes are provided and there is a stack per process per access mode.

2. Protection is hierarchical with the innermost access mode being the least restricted and the outermost the most restricted.

3. Four instructions are provided to change the processor mode to the four access modes (CHMU, CHMS, CHME, and CHMK); furthermore, when a process is executing a change mode instuction the access mode can only be decreased (changed to a more privileged mode) or left the same.

4. Two instructions are provided to validate the accessibility of arguments, Probe Read (PROBER) and Probe Write (PROBEW). These instructions validate the accessibility of arguments using the maximization of the Previous Mode field of PSL and a specified access mode. Thus only current and more restricted access modes can be probed.

5. The Return from Interrupt instruction (REI) insures that the current mode field of the restored PSL is greater than or equal to the current mode field of the current PSL and that the previous mode field of the restored PSL is greater than or equal to the current mode field of the restored PSL.

Given the above operating system and hardware assumptions, the following rules guarantee that less privileged modes cannot pass bogus addresses to more privileged modes.

1. All addresses (including indirect addresses) passed as arguments to an inner access mode must be copied (preferably to a register, but in any case to an area of memory that is not modifiable by less privileged modes) before the accessibility of the actual argument is validated. Furthermore, if such an address will later be used to asynchronously post information back to an outer access mode, then the least privileged access mode that can perform the specified operation (i.e., a read or write of data), must be copied from the coresponding page table entry and stored with the argument address.

   NOTE

   Using least privilege does not work properly when the data structure resides in pages with different protection and the first page has a lesser protection value than the others. When checking the accessibility of such a structure in the context of the serial execution of the process, the check will succeed, but later when the accessibility is checked again during the asynchronous posting of information, the check will fail. This situation is considered to be an operating system bug (may cause the generation of a bug check) and merely causes no information to be posted.

2. The synchronous validation of argument addresses (i.e., as the result of serial program execution) must be explicitly coded using Probe instructions specifying an access mode of zero (i.e., cause maximization to previous access mode).

3. The asynchronous validation of argument addresses (i.e., as the result of software interrupts) must be explicitly coded using Probe instructions specifying the least privileged access mode stored when the argument address was saved (see 1. above) and with a previous access mode field equal to or greater than that of the current mode field of PSL (i.e., cause maximization to least privileged access mode).

4. All arguments to be written must be PROBEW'ed before they are written (protection hole if not).

5. All arguments to be read must be PROBER'ed before they are read to defend against arguments mapped to I/O space and thereby causing an I/O side effect.

6. All addresses passed from an outer access mode to an inner access mode must be copied and validated before being passed as arguments in a call to a more inner access mode. \This insures the integrity of intermediate modes.\

The above discussion centered on the validation of argument addresses. There are other arguments that also deserve the careful handling described. Such arguments are typically address modifiers (e.g., a buffer length) and in most cases must also be copied to insure system integrity.

It is believed that the above assumptions and specified rules for validation make it possible to construct a reliable operating system. No claim is made as to whether a secure operating system can be built.

[End of Appendix J]

Title: VAX-11 Programming Examples -- Rev 4

Specification Status: Fully approved

Architectural Status: under ECO control

File: SRKR4.RNO

PDM #: not used

Date: 24-Mar-77

Superseded Specs:

Author: W. Strecker

Typist: L. Principe

Reviewer(s): P. Conklin, D. Cutler, D. Hustvedt, J. Leonard, P. Lipman, D. Rodgers, S. Rothman, B. Stewart, B. Strecker

Abstract: The examples in Appendix K are designed to illustrate the capabilities of the VAX-11 instruction set. They are not intended to be a tutorial on programming. A familiarity with PDP-11 assembly language programming is assumed. There is no suggestion that a compiler or a programmer might code these examples as production code, rather they are intended as illustrations only.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Distributed	Strecker	25-Sep-75
Rev 2	ECOs 1-11	Strecker	8-Mar-76
Rev 3	ECOs 12-18 and April Meeting	Conklin	13-Jun-76
Rev 4	Add comments; keep with ECOs	Conklin	24-Mar-77

Rev 3 to Rev 4:

1. Add some comments.
2. Track EDITPC, decimal, POLY ECOs.
3. Add EDITPC examples.

Rev 2 to Rev 3:

1. Update assembler constant notation
2. Don't save R0, R1
3. Change to MOVAL, BLEQ, AOBLEQ, AOBLSS, RET
4. Change to 10$
5. Correct loop ending tests
6. Change terminology to post-indexed
7. Add SIN example
8. Add Floating output example
9. Add MOVL of N in SORT example
10. Remove space after comma in examples
11. Add integer/numeric overflow enables to entry masks
12. Add POLY to SIN example

Rev 1 to Rev 2:

1. Remove EXCHx instruction
2. Change to positive arg displacement
3. Remove LSTO
4. Don't precompute the N argument

[End of SRKR4.RNO]

APPENDIX K

PROGRAMMING EXAMPLES

24-Mar-77 -- Rev 4

K.1 PURPOSE

The purpose of the programming examples is to illustrate VAX-11 capabilities which are not present in the PDP-11. It is not intended to be tutorial on programming; a familiarity with PDP-11 assembly language programming is assumed.

K.2 SORT ALGORITHM

The following subroutine written in FORTRAN is an algorithm for sorting an array of values into ascending order.

```
      SUBROUTINE SORT(N,A)
      <data type x> A(N), TEMP
      INTEGER*4 N, I, J
      DO 10 I=1, N-1
      DO 10 J=I+1, N
      IF (A(I).LE.A(J)) GO TO 10
      TEMP = A(I)
      A(I) = A(J)
      A(J) = TEMP
10    CONTINUE
      RETURN
      END
```

The following is VAX-11 code to implement this algorithm. There is no suggestion that any given FORTRAN compiler would generate this code; the algorithm was expressed in FORTRAN only for convenience.

The subroutine is assumed to be called by the VAX-11 standard calling convention (See Appendix C); hence, 4(AP) points to the address of N and 8(AP) points to the address of A (0 origin assumed).

```
        SORT::
 1.             .WORD   ^X400C          ;Entry mask to save R3, R2
                                        ; and enable integer overflow
 2.             MOVAL   @8(AP),R0       ;Get A base
 3.             MOVL    @4(AP),R12      ;Get N (size)
 4.             MOVL    #1,R1           ;Initialize I
 5.      1$:    ADDL3   #1,R1,R2        ;Initialize J to I+1
 6.      2$:    CMPx    (R0)[R1],(R0)[R2] ;Correct order?
 7.             BLEQ    10$             ;Yes
 8.             MOVx    (R0)[R1],R3     ;Save A(I)
 9.             MOVx    (R0)[R2],(R0)[R1] ;Replace A(I) with A(J)
10.             MOVx    R3,(R0)[R2]     ;Replace A(J) with saved A(I)
11.      10$:   AOBLEQ  R12,R2,2$       ;Continue
12.             AOBLSS  R12,R1,1$       ;Continue
13.             RET                     ;Return and restore
                                        ;registers R2 and R3
```

Line 1 contains an entry mask so that registers R2, and R3 will be saved by the CALL instruction which calls the subroutine. By convention, R0 and R1 are not saved. Integer overflow is enabled.

Line 2 gets the base of the A array. The move address instruction is used in conjunction with argument mode addressing. This instruction saves memory accesses inside the loop.

Line 3 gets the array size. The move long instruction is used in conjunction with argument mode addressing. This instruction saves memory accesses inside the loop.

Line 4 initializes I to 1. Literal mode addressing is used.

Line 5 initializes J with I+1. A three operand add is used.

Line 6 compares A(I) to A(J). Register post-indexed mode addressing is used.

Line 7 branches past the exchange if the array elements are in the right order.

Lines 8 through 10 exchange the array elements if they are in the wrong order. Register post-indexed mode addressing is used.

Lines 11 and 12 carry out the loop end operations. Argument mode addressing is used.

Line 13 returns and restores registers R2 and R3.

Note, that because of logical indexing in Lines 5, 7, 8, and 9 and the orthogonality of operator and data type, the subroutine works for byte, word, longword, floating, or double data types of array A simply by substituting B, W, L, F, or D respectively for x. Note that if double, then R4 would have to be saved also in the entry mask.

The size of each instruction is:

1.	2 bytes
2.	4
3.	4
4.	3
5.	4
6.	5
7.	2
8.	4
9.	5
10.	4
11.	4
12.	4
13.	1
Total	46 bytes

K.3 SIN FUNCTION

This example shows how the initial argument handling might be done in the math library to handle argument range reduction followed by CASEing to the algorithm for each octant.

```
        ;
        ; X = SIN (Y)
        ;

        PIHI=xxx                        ;high 4 bytes (8 if double)
        PILO=xxx                        ;low byte of 4/PI

        SIN::
                .WORD   ^X400C          ;save R2-R3 for POLYF, -R7 for POLYD
                                        ;enable integer overflow
                MOVAL   HANDLER,0(FP)   ;enable integer overflow
                                        ; condition handler to catch
                                        ; loss of significance on
                                        ; a huge argument
                EMODx   #PIHI,#PILO,@4(AP),R2,R0
                                        ;get octant in R2
                                        ; reduced argument in R0
                BGEQ    1$              ;if positive, ok
                ADDx    #^F1.0,R0       ;if negative, get
                DECL    R2              ; positive reduction
        1$:     BICB2   #^C7,R2         ;mask to 8 octants
                CASEB   R2,#1,#6        ;branch to each octant
        2$:     .WORD   OCT_1-2$
                .WORD   OCT_2-2$
                .WORD   OCT_3-2$
                .WORD   OCT_4-2$
                .WORD   OCT_5-2$
                .WORD   OCT_6-2$
                .WORD   OCT_7-2$
                                        ;fall out of CASE on octant 0

        ;
        ; octant 0 with fully precise reduced argument in R0
        ;

        OCT_0:  POLYx   R0,2$,1$        ;evaluate polynomial
                RET                     ;return value in R0

        1$:     .FLOAT  ...
                .FLOAT  ...
                ...
        2$=.-1$-1
                ...

        HANDLER:                        ;condition handler
                .WORD   ...
                ...
```

K.4 FIXED FORMAT FLOATING OUTPUT

This example shows how to output a floating point number in the FORTRAN format F9.3.

```
        ;
        ; string = FOUT (X)
        ;

        STRING: .BLKB   10              ;room for output

        PATTERN:                        ;EDITPC pattern string
                EO$FLOAT        4       ;float sign, move 4 digits
                EO$END_FLOAT            ;end floating sign
                EO$MOVE         1       ;move one digit
                EO$INSERT       ^A/./   ;insert period
                EO$MOVE         3       ;move three fractional digits
                EO$END                  ;end of pattern

        FOUT::
                .WORD   ^XC03C          ;save R2-R5, enable overflows
                SUBL2   #8,SP           ;make room on stack
                MULF3   #^F1000.0,@4(AP),R0
                                        ;normalize for the .3
                CVTRFL  R0,R0           ;round digits
                CVTLP   R0,#8,(SP)      ;convert to digits on stack
                EDITPC  #8,(SP),PATTERN,STRING
                                        ;edit to output
                MOVQ    #<.LONG 9,STRING+1>,R0
                                        ;function value is a
                                        ; string descriptor
                RET                     ;return restoring R2-R5
                                        ; and the stack
```

K.5 COBOL OUTPUT EDITING

In all of these examples, A is a COMP-3 datum of length A_LEN. The operation is

```
MOVE A TO B.
```

The generated code is

```
EDITPC  #A_LEN,@A,MICRO,@B
```

In the patterns, the EO$ADJUST_INPUT can be omitted if A is the same size as B, and the EO$REPLACE_SIGN (and its EO$LOAD_FILL) can be omitted if A cannot contain a -0.

1. PICTURE $$,$$9.99CR

```
MICRO:  EO$ADJUST_INPUT         6
        EO$LOAD_SIGN            '$
        EO$FLOAT                1
        EO$INSERT               ',
        EO$FLOAT                2
        EO$END_FLOAT
        EO$MOVE                 1
        EO$INSERT               '.
        EO$MOVE                 2
        EO$LOAD_PLUS            '
        EO$LOAD_MINUS           'C
        EO$STORE_SIGN
        EO$LOAD_MINUS           'R
        EO$STORE_SIGN
        EO$REPLACE_SIGN         2
        EO$REPLACE_SIGN         1
        EO$END
```

2. PICTURE +$99,999.99

```
MICRO:  EO$ADJUST_INPUT         7
        EO$LOAD_PLUS            '+
        EO$STORE_SIGN
        EO$SET_SIGNIF
        EO$INSERT               '$
        EO$MOVE                 2
        EO$INSERT               ',
        EO$MOVE                 3
        EO$INSERT               '.
        EO$MOVE                 2
        EO$LOAD_FILL            '+
        EO$REPLACE_SIGN         11
        EO$END
```

3. PICTURE ZZ,ZZZ.ZZ

```
MICRO:   EO$ADJUST_INPUT          7
         EO$MOVE                  2
         EO$INSERT                ',
         EO$MOVE                  3
         EO$SET_SIGNIF
         EO$INSERT                '.
         EO$MOVE                  2
         EO$BLANK_ZERO            3
         EO$END
```

4. PICTURE 99,999.99 BLANK WHEN ZERO

```
MICRO:   EO$ADJUST_INPUT          7
         EO$SET_SIGNIF
         EO$MOVE                  2
         EO$INSERT                ',
         EO$MOVE                  3
         EO$INSERT                '.
         EO#MOVE                  2
         EO$BLANK_ZERO            9
         EO$END
```

5. PICTURE -----9.99

```
MICRO:   EO$ADJUST_INPUT          7
         EO$FLOAT                 4
         EO$END_FLOAT
         EO$MOVE                  1
         EO$INSERT                '.
         EO$MOVE                  2
         EO$REPLACE_SIGN          5
         EO$END
```

6. PICTURE +++++9.99

```
MICRO:   EO$ADJUST_INPUT          7
         EO$LOAD_PLUS             '+
         EO$FLOAT                 4
         EO$END_FLOAT
         EO$MOVE                  1
         EO$INSERT                '.
         EO$MOVE                  2
         EO$LOAD_FILL             '+
         EO$REPLACE_SIGN          5
         EO$END
```

7. PICTURE **,***.**

```
     MICRO:    EO$ADJUST_INPUT          7
               EO$LOAD_FILL             '*
               EO$MOVE                  2
               EO$INSERT                ',
               EO$MOVE                  3
               EO$SET_SIGNIF
               EO$INSERT                ',
               EO$MOVE                  2
               EO$BLANK_ZERO            2
               EO$END
```

8. PICTURE BBBZZBZZZ.ZZB

```
     MICRO:    EO$ADJUST_INPUT          7
               EO$FILL                  3
               EO$MOVE                  2
               EO$FILL                  1
               EO$MOVE                  3
               EO$SET_SIGNIF
               EO$INSERT                '.
               EO$MOVE                  2
               EO$BLANK_ZERO            3
               EO$FILL                  1
               EO$END
```

[End of Appendix K]

INDEX