VAX-11
Architecture Reference Manual

20 May 1982
Revision 6.1
The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The specifications, herein, are the property of Digital Equipment Corporation and shall not be reproduced or copied in whole or in part as the basis for the manufacture or sale of items without written permission.

Copyright (c) 1976, 1977, 1979, 1980, 1981 by Digital Equipment Corporation

The following are trademarks of Digital Equipment Corporation:

ASSIST-11 DIBOL K110 RSTS
COMPUTER LABS DIGITAL KL10 RGX
COMSYST DNC LAB-8 RT-11
COMTEX EDGRIN LAB-K RTS-8
DDT EDUSYSTEM MASSBUS SABR
DEC FLIP CHIP OMNIBUS SBI
DECnet FOCAL OS/8 TRAX
DECCOMM GLC-8 PDP TYPESET-8
DECSUS IDAC PHA TYPESET-10
DECSYSTEM-10 IDACS PS/8 TYPESET-11
DECSYSTEM-20 INDAC QUICKPOINT UNIBUS
DECTape KA10 RAD-8 VAX
PREFACE

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

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

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

## CHAPTER 1

- **INTRODUCTION**
  - 1.1 INTRODUCTION .......................................................... 1-1
  - 1.2 TERMINOLOGY AND CONVENTIONS .................................... 1-2
    - 1.2.1 Numbering .......................................................... 1-2
    - 1.2.2 UNPREDICTABLE And UNDEFINED ............................... 1-2
    - 1.2.3 Ranges And Extents .............................................. 1-2
    - 1.2.4 MBZ ................................................................. 1-3
    - 1.2.5 Reserved ............................................................ 1-3
    - 1.2.6 Figure Drawing Conventions .................................... 1-3

## CHAPTER 2

- **BASIC ARCHITECTURE**
  - 2.1 ADDRESSING ........................................................... 2-1
  - 2.2 DATA TYPES ........................................................... 2-1
    - 2.2.1 Byte .............................................................. 2-1
    - 2.2.2 Word ............................................................. 2-2
    - 2.2.3 Longword ......................................................... 2-2
    - 2.2.4 Quadword ........................................................ 2-3
    - 2.2.5 Octaword .......................................................... 2-3
    - 2.2.6 F floating ........................................................ 2-4
    - 2.2.7 D_{floating} ...................................................... 2-4
    - 2.2.8 G_{floating} ...................................................... 2-5
    - 2.2.9 H_{floating} ...................................................... 2-5
    - 2.2.10 Variable Length Bit Field .................................... 2-6
    - 2.2.11 Character String .............................................. 2-8
    - 2.2.12 Trailing Numeric String ..................................... 2-8
    - 2.2.13 Leading Separate Numeric String ............................ 2-11
    - 2.2.14 Packed Decimal String ....................................... 2-13
  - 2.3 PROCESSOR STATE ..................................................... 2-15
  - 2.4 PROCESSOR STATUS WORD ........................................... 2-17
    - 2.4.1 C Bit ............................................................ 2-17
    - 2.4.2 V Bit ............................................................ 2-17
    - 2.4.3 Z Bit ............................................................ 2-17
    - 2.4.4 N Bit ............................................................ 2-17
    - 2.4.5 T Bit ............................................................ 2-18
    - 2.4.6 IV Bit ........................................................... 2-18
    - 2.4.7 FU Bit ........................................................... 2-18
    - 2.4.8 DV Bit ........................................................... 2-18
  - 2.5 PERMANENT EXCEPTION ENABLES ..................................... 2-19
    - 2.5.1 Divide By Zero ................................................. 2-19
    - 2.5.2 Floating Overflow .............................................. 2-19
  - 2.6 INSTRUCTION FORMAT ............................................... 2-19
  - 2.7 SEPARATION OF PROCEDURE AND DATA .............................. 2-20
  - 2.8 I/O STRUCTURE ....................................................... 2-20
  - 2.9 INTERRUPT STRUCTURE ............................................... 2-20

## CHAPTER 3

- **INSTRUCTION FORMATS AND ADDRESSING MODES**
  - 3.1 OPCODE FORMATS ...................................................... 3-1
  - 3.2 OPERAND SPECIFIERS ................................................. 3-2
  - 3.3 NOTATION .............................................................. 3-4
  - 3.4 GENERAL MODE ADDRESSING FORMATS ............................... 3-5
CHAPTER 4

INSTRUCTIONS

4.1 INSTRUCTION SET ............................................. 4-1
4.1.1 Instruction Descriptions ................................ 4-1
4.1.2 Operand Specifier Notation ............................. 4-3
4.1.3 Operation Description Notation ........................ 4-4
4.2 INTEGER ARITHMETIC AND LOGICAL INSTRUCTIONS ...... 4-7
4.3 ADDRESS INSTRUCTIONS .................................... 4-37
4.4 VARIABLE LENGTH BIT FIELD INSTRUCTIONS ........... 4-40
4.5 CONTROL INSTRUCTIONS ................................... 4-48
4.6 PROCEDURE CALL INSTRUCTIONS ........................... 4-70
4.7 MISCELLANEOUS INSTRUCTIONS ............................. 4-78
4.8 QUEUE INSTRUCTIONS ...................................... 4-90
4.8.1 Absolute Queues ........................................ 4-90
4.8.2 Self-relative Queues .................................... 4-95
4.8.3 Instruction Descriptions ................................. 4-98
4.9 FLOATING POINT INSTRUCTIONS ........................... 4-115
4.9.1 Introduction ............................................. 4-115
4.9.2 Overview Of The Instruction Set ....................... 4-117
4.9.3 Accuracy ................................................ 4-117
4.9.4 Instruction Descriptions ................................. 4-120
4.10 CHARACTER STRING INSTRUCTIONS ...................... 4-146
4.11 CYCLIC REDUNDANCY CHECK INSTRUCTION ............... 4-171
4.12 DECIMAL STRING INSTRUCTIONS .......................... 4-175
4.12.1 Decimal Overflow ...................................... 4-176
4.12.2 Zero Numbers .......................................... 4-176
4.12.3 Reserved Operand Exception .......................... 4-176
4.12.4 UNPREDICTABLE Results .............................. 4-176
4.12.5 Packed Decimal Operations ........................... 4-177
4.12.6 Zero Length Decimal Strings ......................... 4-177
4.12.7 Instruction Descriptions ............................... 4-178
4.13 EDIT INSTRUCTION ........................................ 4-205
4.14 OTHER VAX-11 INSTRUCTIONS ............................. 4-226

CHAPTER 5

MEMORY MANAGEMENT

5.1 INTRODUCTION .............................................. 5-1
5.2 VIRTUAL ADDRESS SPACE .................................. 5-2
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>Process Space</td>
<td>5-4</td>
</tr>
<tr>
<td>5.2.2</td>
<td>System Space</td>
<td>5-4</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Virtual Address Format</td>
<td>5-4</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Virtual Address Space Layout</td>
<td>5-5</td>
</tr>
<tr>
<td>5.3</td>
<td>MEMORY MANAGEMENT CONTROL</td>
<td>5-5</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Memory Management Disabled</td>
<td>5-5</td>
</tr>
<tr>
<td>5.4</td>
<td>ADDRESS TRANSLATION</td>
<td>5-6</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Page Table Entry (PTE)</td>
<td>5-6</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Page Table Entry (PTE) For I/O Devices</td>
<td>5-8</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Changes To Page Table Entries</td>
<td>5-9</td>
</tr>
<tr>
<td>5.5</td>
<td>ACCESS CONTROL</td>
<td>5-10</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Processor Modes</td>
<td>5-10</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Protection Code</td>
<td>5-10</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Length Violation</td>
<td>5-13</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Access Control Violation Fault</td>
<td>5-13</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Access Across A Page Boundary</td>
<td>5-13</td>
</tr>
<tr>
<td>5.5.6</td>
<td>System Space Address Translation</td>
<td>5-13</td>
</tr>
<tr>
<td>5.5.7</td>
<td>Process Space Address Translation</td>
<td>5-16</td>
</tr>
<tr>
<td>5.5.8</td>
<td>P0 Region</td>
<td>5-17</td>
</tr>
<tr>
<td>5.5.9</td>
<td>P1 Region</td>
<td>5-20</td>
</tr>
<tr>
<td>5.6</td>
<td>TRANSLATION BUFFER</td>
<td>5-22</td>
</tr>
<tr>
<td>5.7</td>
<td>FAULTS AND PARAMETERS</td>
<td>5-23</td>
</tr>
<tr>
<td>5.8</td>
<td>PRIVILEGED SERVICES AND ARGUMENT VALIDATION</td>
<td>5-24</td>
</tr>
<tr>
<td>5.8.1</td>
<td>Changing Access Modes</td>
<td>5-24</td>
</tr>
<tr>
<td>5.8.2</td>
<td>Validating Address Arguments (PROBE instructions)</td>
<td>5-25</td>
</tr>
<tr>
<td>5.8.3</td>
<td>Notes On The PROBE instructions</td>
<td>5-28</td>
</tr>
</tbody>
</table>

**CHAPTER 6**

**EXCEPTIONS AND INTERRUPTS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>INTRODUCTION</td>
<td>6-1</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Processor Interrupt Priority Levels (IPL)</td>
<td>6-2</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Interrupts</td>
<td>6-2</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Exceptions</td>
<td>6-3</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Contrast Between Exceptions And Interrupts</td>
<td>6-3</td>
</tr>
<tr>
<td>6.2</td>
<td>PROCESSOR STATUS</td>
<td>6-5</td>
</tr>
<tr>
<td>6.3</td>
<td>INTERRUPTS</td>
<td>5-8</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Urgent Interrupts -- Levels 18-1F (Hex)</td>
<td>6-9</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Device Interrupts -- Levels 10-17 (Hex)</td>
<td>6-9</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Software Generated Interrupts -- Levels 01-0F (Hex)</td>
<td>6-10</td>
</tr>
<tr>
<td>6.3.3.1</td>
<td>Software Interrupt Summary Register</td>
<td>6-10</td>
</tr>
<tr>
<td>6.3.3.2</td>
<td>Software Interrupt Request Register</td>
<td>6-10</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Interrupt Priority Level Register</td>
<td>6-11</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Interrupt Example</td>
<td>6-12</td>
</tr>
<tr>
<td>6.4</td>
<td>EXCEPTIONS</td>
<td>5-13</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Arithmetic Traps/Faults</td>
<td>6-14</td>
</tr>
<tr>
<td>6.4.1.1</td>
<td>Integer Overflow Trap</td>
<td>6-14</td>
</tr>
<tr>
<td>6.4.1.2</td>
<td>Integer Divide By Zero Trap</td>
<td>6-15</td>
</tr>
<tr>
<td>6.4.1.3</td>
<td>Floating Overflow Trap</td>
<td>6-15</td>
</tr>
<tr>
<td>6.4.1.4</td>
<td>Divide By Zero Trap – Floating or Decimal String</td>
<td>6-15</td>
</tr>
<tr>
<td>6.4.1.5</td>
<td>Floating Underflow Trap</td>
<td>6-15</td>
</tr>
<tr>
<td>6.4.1.6</td>
<td>Decimal String Overflow Trap</td>
<td>6-15</td>
</tr>
</tbody>
</table>
CHAPTER 9

PRIVILEGED REGISTERS

9.1 PROCESSOR REGISTER SPACE ........................................ 9-1
9.2 PER-PROCESS REGISTERS AND CONTEXT SWITCHING ................ 9-1
9.3 STACK POINTER IMAGES .............................................. 9-2
9.4 MTPR AND MPPR INSTRUCTIONS .................................... 9-3
9.5 VAX-11 SERIES REGISTERS .......................................... 9-5
9.5.1 System Identification Register (SID) ................................. 9-7
9.5.2 Console Terminal Registers ........................................ 9-8
9.5.2.1 VAX-11/780 console register implementation ................ 9-9
9.5.2.1.1 Status Byte Definition .................................... 9-11
9.5.3 Clock Registers ..................................................... 9-11
9.5.3.1 Time-of-Year Clock ............................................ 9-11
9.5.3.2 Interval Clock .................................................. 9-13
9.6 VAX-11/780 SPECIFIC REGISTERS .................................. 9-15
9.6.1 VAX-11/780 Accelerator .......................................... 9-15
9.6.2 VAX-11/780 Micro Control Store ................................ 9-17
9.6.3 SBI FAULT/STATUS REGISTER (SBIFS) ............................ 9-18
9.6.4 SBI SILO DATA REGISTER (SBIS) ................................ 9-19
9.6.5 SBI SILO COMPARATOR REGISTER (SBISC) ....................... 9-19
9.6.6 SBI MAINTENANCE REGISTER (SBIMT) ............................. 9-20
9.6.7 SBI ERROR REGISTER (SBIER) .................................... 9-21
9.6.8 SBI TIMEOUT ADDRESS (SBITA) .................................. 9-22
9.6.9 SBI QUAD CLEAR (SBIQC) ......................................... 9-23
9.7 VAX-11/750 SPECIFIC REGISTERS .................................. 9-24
9.7.1 CMI Error Register .................................................. 9-24
9.7.2 Console Storage Device Registers ................................. 9-25
9.7.3 Translation Buffer Group Disable Register (TBDR) ............. 9-26
9.7.4 Cache Disable Register (CADR) ................................... 9-26
9.7.5 Machine Check Error Summary Register (MCESR) ............... 9-26
9.7.6 Cache Error Register (CAER) ..................................... 9-26
9.7.7 Accelerator Control/Status Register .............................. 9-27
9.7.8 Initialize UNIBUS (IORESET) ..................................... 9-27
9.7.9 Translation Buffer Data Register (TBDATA) ..................... 9-28

CHAPTER 10

PDP-11 COMPATIBILITY MODE

10.1 INTRODUCTION ..................................................... 10-1
10.2 COMPATIBILITY MODE USER ENVIRONMENT .......................... 10-2
10.2.1 General Registers And Addressing Modes ....................... 10-2
10.2.1.1 Register Mode ................................................ 10-2
10.2.1.2 Register Deferred Mode ..................................... 10-2
10.2.1.3 Autoincrement Mode ......................................... 10-3
10.2.1.4 Autodecrement Deferred Mode ................................. 10-3
10.2.1.5 Autodecrement Mode ......................................... 10-4
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2.1.6</td>
<td>Autodecrement Deferred Mode</td>
<td>10-4</td>
</tr>
<tr>
<td>10.2.1.7</td>
<td>Index Mode</td>
<td>10-5</td>
</tr>
<tr>
<td>10.2.1.8</td>
<td>Index Deferred Mode</td>
<td>10-5</td>
</tr>
<tr>
<td>10.2.2</td>
<td>The Stack</td>
<td>10-5</td>
</tr>
<tr>
<td>10.2.3</td>
<td>Processor Status Word</td>
<td>10-6</td>
</tr>
<tr>
<td>10.2.4</td>
<td>Instructions</td>
<td>10-7</td>
</tr>
<tr>
<td>10.2.4.1</td>
<td>Single Operand Instructions</td>
<td>10-9</td>
</tr>
<tr>
<td>10.2.4.2</td>
<td>Double Operand Instructions</td>
<td>10-24</td>
</tr>
<tr>
<td>10.2.4.3</td>
<td>Branch Instructions</td>
<td>10-36</td>
</tr>
<tr>
<td>10.2.4.4</td>
<td>Jump And Subroutine Instructions</td>
<td>10-42</td>
</tr>
<tr>
<td>10.2.4.5</td>
<td>Return From Interrupts And Traps</td>
<td>10-45</td>
</tr>
<tr>
<td>10.2.4.6</td>
<td>Miscellaneous</td>
<td>10-48</td>
</tr>
<tr>
<td>10.3</td>
<td>ENTERING AND LEAVING COMPATIBILITY MODE</td>
<td>10-53</td>
</tr>
<tr>
<td>10.3.1</td>
<td>General Register Usage</td>
<td>10-53</td>
</tr>
<tr>
<td>10.4</td>
<td>COMPATIBILITY MODE MEMORY MANAGEMENT</td>
<td>10-54</td>
</tr>
<tr>
<td>10.5</td>
<td>COMPATIBILITY MODE EXCEPTIONS AND INTERRUPTS</td>
<td>10-57</td>
</tr>
<tr>
<td>10.5.1</td>
<td>Reserved Instruction Fault</td>
<td>10-57</td>
</tr>
<tr>
<td>10.5.2</td>
<td>BPT Instruction Fault</td>
<td>10-57</td>
</tr>
<tr>
<td>10.5.3</td>
<td>IOT Instruction Fault</td>
<td>10-57</td>
</tr>
<tr>
<td>10.5.4</td>
<td>EMT Instruction Fault</td>
<td>10-57</td>
</tr>
<tr>
<td>10.5.5</td>
<td>TRAP Instruction Fault</td>
<td>10-57</td>
</tr>
<tr>
<td>10.5.6</td>
<td>Illegal Instruction Fault</td>
<td>10-57</td>
</tr>
<tr>
<td>10.5.7</td>
<td>Odd Address Error Abort</td>
<td>10-58</td>
</tr>
<tr>
<td>10.6</td>
<td>T BIT OPERATION IN COMPATIBILITY MODE</td>
<td>10-58</td>
</tr>
<tr>
<td>10.7</td>
<td>UNIMPLEMENTED PDP-11 TRAPS</td>
<td>10-60</td>
</tr>
<tr>
<td>10.8</td>
<td>COMPATIBILITY MODE I/O REFERENCES</td>
<td>10-61</td>
</tr>
<tr>
<td>10.9</td>
<td>PROCESSOR REGISTERS</td>
<td>10-61</td>
</tr>
<tr>
<td>10.10</td>
<td>PROGRAM SYNCHRONIZATION</td>
<td>10-61</td>
</tr>
</tbody>
</table>

**APPENDIX A**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>INSTRUCTION OPERAND FORMATS</td>
<td>A-1</td>
</tr>
<tr>
<td>A.2</td>
<td>OPERAND SPECIFIER NOTATION</td>
<td>A-9</td>
</tr>
<tr>
<td>A.3</td>
<td>OPCODE ASSIGNMENTS</td>
<td>A-12</td>
</tr>
<tr>
<td>A.4</td>
<td>INSTRUCTIONS USABLE TO REFERENCE I/O SPACE</td>
<td>A-18</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1-Feb-80 -- Rev 6

1.1 INTRODUCTION

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

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

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

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

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

3. A systematic, elegant instruction set with orthogonality of operators, data types, and addressing modes. This enables the instruction set to be exploited easily, particularly by high level language processors.
4. Extensibility. The instruction set is designed so that new data types and operators can be included efficiently in a manner consistent with the currently defined operators and data types.

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

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

1.2 TERMINOLOGY AND CONVENTIONS

1.2.1 Numbering

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

1.2.2 UNPREDICTABLE And UNDEFINED

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

1.2.3 Ranges And Extents

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

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

1.2.5 Reserved

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

1.2.6 Figure Drawing Conventions

Figures which depict registers or memory follow the convention that increasing addresses run right to left and top to bottom.
CHAPTER 2
BASIC ARCHITECTURE

29-Feb-80 -- Rev 6

2.1 ADDRESSING

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

2.2 DATA TYPES

2.2.1 Byte

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

```
             7  0
          +----------------+ :A
          |                |
          +----------------+
```

A byte is specified by its address A. When interpreted arithmetically, a byte is a twos complement integer with bits of increasing significance going 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:

```
1 0
+-----------------------------+ 1 :A
|                            |
+-----------------------------+
```

A word is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a word is a two's 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:

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

A longword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a longword is a two's 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 31 through 16 and 0 through 14. Bit 31 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:

```
 3
 1  +----------------+ 0
   |                |
   |                |
   +----------------+ :A
   |                | :A+4
   +----------------+ :A+8
   |                | :A+12
   +----------------+
   |                |
   |                |
   +----------------+
   |                |
   +----------------+ 6
 3
 2
```

A quadword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a quadword is a two's 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.

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:

```
 3
 1  +----------------+ 0
   |                |
   |                |
   +----------------+ :A
   |                | :A+4
   +----------------+ :A+8
   |                | :A+12
   +----------------+ 9
   |                |
   |                |
   +----------------+ 6
 2
 1
```

A octaword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, an octaword is a two's 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.
2.2.6 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.

```
  1 1 5 4                     7 5 0
  +---------------------------+
 |S| exp | fraction | :A
  +---------------------------+
 | fraction | :A+2
  +---------------------------+
```

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 15 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 1, 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 0.29*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 D_floating

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

```
  1 1 5 4                     7 6 0
  +---------------------------+
 |S| exp | fraction | :A
  +---------------------------+
 | fraction | :A+2
  +---------------------------+
 | fraction | :A+4
  +---------------------------+
 | fraction | :A+6
  +---------------------------+
```

A D_floating datum is specified by its address A, the address of the byte containing bit 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:

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

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

2.2.9 H_floating

A H_floating datum is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 127:
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 53, 32 through 47, and 15 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 \( .84 \times 10^{4932} \) through \( .59 \times 10^{4932} \). The precision of a H floating datum is approximately one part in \( 2^{*112} \), i.e., typically 33 decimal digits.

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.
For bit strings in memory, the position is in the range \(-2^{31}\) through \(2^{31}-1\) and is conveniently viewed as a signed 29-bit byte offset and a 3-bit bit-within-byte field:

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

The sign extended 29-bit byte offset is added to the address \(A\) and the resulting address specifies the byte in which the field begins. The 3-bit bit-within-byte field encodes the starting position (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 two's 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.

For bit fields in registers, the position is in the range 0 through 31. The position operand specifies the starting position (0 through 31) of the field in the register. A variable bit field may be contained in 2 registers if the sum of position and size exceeds 32.

```
  3 2 1
+-------------------------+---------+
|                       | R[n+1]  |
+-------------------------+---------+
```

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

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

```
    7   0
   +-------+-
  |       | :A
   +-------+-
     .     .
   +-------+-
  |       | :A+L-1
   +-------+-
    7   0
```

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

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

The length L of a string is in the range 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:
<table>
<thead>
<tr>
<th>digit</th>
<th>decimal</th>
<th>hex</th>
<th>ASCII character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>39</td>
<td>9</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Digit</th>
<th>Zoned Numeric Format</th>
<th>Overpunch Format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>decimal</td>
<td>hex</td>
</tr>
<tr>
<td>0</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>39</td>
</tr>
<tr>
<td>-0</td>
<td>112</td>
<td>70</td>
</tr>
<tr>
<td>-1</td>
<td>113</td>
<td>71</td>
</tr>
<tr>
<td>-2</td>
<td>114</td>
<td>72</td>
</tr>
<tr>
<td>-3</td>
<td>115</td>
<td>73</td>
</tr>
<tr>
<td>-4</td>
<td>116</td>
<td>74</td>
</tr>
<tr>
<td>-5</td>
<td>117</td>
<td>75</td>
</tr>
<tr>
<td>-6</td>
<td>118</td>
<td>76</td>
</tr>
<tr>
<td>-7</td>
<td>119</td>
<td>77</td>
</tr>
<tr>
<td>-8</td>
<td>120</td>
<td>78</td>
</tr>
<tr>
<td>-9</td>
<td>121</td>
<td>79</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>: A</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>: A+1</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>: A+2</td>
<td></td>
</tr>
</tbody>
</table>

Overpunch Format

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>: A</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>: A+1</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>: A+2</td>
<td></td>
</tr>
</tbody>
</table>

and "-123" is represented:

Zoned Format

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>: A</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>: A+1</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>: A+2</td>
<td></td>
</tr>
</tbody>
</table>

Overpunch Format

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>: A</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>: A+1</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>: A+2</td>
<td></td>
</tr>
</tbody>
</table>

2.2.13 Leading Separate Numeric String

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

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

<table>
<thead>
<tr>
<th>Sign</th>
<th>decimal</th>
<th>hex</th>
<th>ASCII character</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>43</td>
<td>2B</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>32</td>
<td>20</td>
<td>&lt;blank&gt;</td>
</tr>
<tr>
<td>-</td>
<td>45</td>
<td>2D</td>
<td></td>
</tr>
</tbody>
</table>

The preferred representation for "+" is ASCII "+". All subsequent bytes contain an ASCII digit character:
<table>
<thead>
<tr>
<th>digit</th>
<th>decimal</th>
<th>hex</th>
<th>ASCII character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>39</td>
<td>9</td>
</tr>
</tbody>
</table>

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:

```
<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A+3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

and "-123" is:

```
<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A+3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
2.2.14 Packed Decimal String

A packed decimal string is a contiguous sequence of bytes in memory. A packed decimal string is specified by 2 attributes: the address A of the first byte of the string and a length L which is the number of digits in the string and NOT the length of the string in bytes. The bytes of a packed decimal string are divided into 2 4-bit fields (nibbles) which must contain decimal digits except the low nibble (bits 3:0) of the last (highest addressed) byte which must contain a sign. The representation for the digits and sign is:

<table>
<thead>
<tr>
<th>digit or sign</th>
<th>decimal</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>+</td>
<td>10,12,14 or 15</td>
<td>A, C, E, or F</td>
</tr>
<tr>
<td>-</td>
<td>11 or 13</td>
<td>B, or D</td>
</tr>
</tbody>
</table>

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:

```
          7 4 3 0
+-----------+-------
| 1 | 2 | : A
+-----------+-------
| 3 | 12 | : A + 1
+-----------+-------
```

and "-12" has length 2 and is represented:
Basic Architecture
DATA TYPES

| 7 | 4 | 3 | 0 |
+---+---+---+---+
<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>: A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>13</td>
<td>: A + 1</td>
</tr>
</tbody>
</table>
+---+---+---+---+
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, 9, and 7.

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

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

```
  3
  1
+----------------------------------+ 0
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
 +----------------------------------+
     \:Rn
```

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

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

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

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

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

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

When a datum of type byte, word, longword, or F_floating is stored in a register, the bit numbering in the register corresponds to the numbering
Basic Architecture
PROCESSOR STATE

29-Feb-80 -- Rev 6
Page 2-16

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
to 31. As for quadword, D_floating, and G_floating, a pair of
registers R[n] and R[n+1] is treated as a 64-bit register with bits 31:0
in register R[n] and bit 63:32 in register R[n+1].

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

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

```
<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>-------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>F</td>
<td>I</td>
<td></td>
<td>L</td>
<td>V</td>
<td>N</td>
<td>Z</td>
<td>C</td>
</tr>
<tr>
<td>-------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
```

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

2.4.1 C Bit

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

2.4.2 V Bit

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

2.4.3 Z Bit

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

2.4.4 N Bit

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

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

2.4.6 IV Bit

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

2.4.7 FU Bit

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

2.4.8 DV Bit

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

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

2.5.1 Divide By Zero

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

2.5.2 Floating Overflow

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

2.6 INSTRUCTION FORMAT

VAX-11 has a variable length instruction format. An instruction specifies an operation and 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 A for a summary of all operands, instructions, and their binary assignments.
2.7 SEPARATION OF PROCEDURE AND DATA

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

2.8 I/O STRUCTURE

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

2.9 INTERRUPT STRUCTURE

A VAX-11 processor provides a 32 level vectored priority interrupt system. This is described in detail in Chapter 6.
CHAPTER 3
INSTRUCTION FORMATS AND ADDRESSING MINDS

5-May-80 -- Rev 7

3.1 OPCODE FORMATS

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

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>:A</td>
</tr>
<tr>
<td>+---------+</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>opcode</td>
</tr>
<tr>
<td>+---------+-----------------+</td>
</tr>
</tbody>
</table>
3.2 OPERAND SPECIFIERS

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

The access types include:

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

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

The data types include:

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

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

1. Address - the operand size to be used in the address calculation in autoincrement, autodecrement, and index modes.
2. Branch - the size of the branch displacement.
3.3 NOTATION

To describe the addressing modes the following is used:

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

NOTE

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

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

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

3.4.1 Register Mode

The operand specifier format is:

```
7  4  3  0
+-----------+
| 5 | Rn |
+-----------+
```

No specifier extension follows.

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

- `operand = Rn` if one register
- `operand = R[n+1]'Rn` if two registers
- `operand = 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 for register mode is Rn.
3.4.2 Register Deferred Mode

The operand specifier format is:

\[
\begin{array}{cccc}
7 & 4 & 3 & 0 \\
\hline
\text{Rn} & | \\
\end{array}
\]

No specifier extension follows.

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

\[OA = Rn\]
\[\text{operand} = (OA)\]

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

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

3.4.3 Autoincrement Mode

The operand specifier format is:

\[
\begin{array}{cccc}
7 & 4 & 3 & 0 \\
\hline
\text{Rn} & | \\
\end{array}
\]

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

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

\[OA = Rn\]
\[Rn \leftarrow Rn + \text{size}\]
\[\text{operand} = (OA)\]

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

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

3.4.4 Autoincrement Deferred Mode

The operand specifier format is:

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

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

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

```
OA = (Rn)
Rn <- Rn + 4
operand = (OA)
```

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

The operand specifier format is:

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

No specifier extension follows.

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

```
Rn ← Rn - size
OA = Rn
operand = (OA)
```

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

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

There are 3 operand specifier formats:

\[
\begin{array}{c}
7 & 4 & 3 & 0 \\
\hline
1. & 10 & \text{Rn} & \hline
\hline
2. & 12 & \text{Rn} & \hline
\hline
3. & 14 & \text{Rn} & \hline
\end{array}
\]

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

\[
\begin{array}{c}
7 & 4 & 3 & 0 \\
\hline
1. & 12 & \text{Rn} & \hline
\hline
2. & 14 & \text{Rn} & \hline
\hline
3. & 16 & \text{Rn} & \hline
\end{array}
\]

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

\[
\begin{array}{c}
7 & 4 & 3 & 0 \\
\hline
1. & 16 & \text{Rn} & \hline
\hline
2. & 18 & \text{Rn} & \hline
\hline
3. & 20 & \text{Rn} & \hline
\end{array}
\]

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

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

\[
\text{OA} = \text{Rn} + \text{SEXT}(\text{displ}) \quad \text{!if byte or word displacement}
\]
\[
\text{or} \quad \text{Rn} + \text{displ} \quad \text{!if longword displacement}
\]

\[
\text{operand} = (\text{OA})
\]

If \(\text{Rn}\) denotes \(\text{PC}\), the updated contents of \(\text{PC}\) is used. The updated contents of \(\text{PC}\) is the address of the first byte beyond the specifier extension.

The assembler notation for byte, word, and long displacement mode is \(B^D(\text{Rn})\), \(W^D(\text{Rn})\), and \(L^D(\text{Rn})\) respectively where \(D = \text{displ}\).
3.4.7 Displacement Deferred Mode

There are 3 operand specifier formats:

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

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

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

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

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

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

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

\[
OA = (Rn + SEXT(displ)) \quad \text{if byte or word displacement}
\]

\[
(Rn + displ) \quad \text{if longword displacement}
\]

\[
\text{operand} = (OA)
\]

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

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

The operand specifier format is:

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

No specifier extension follows.

For operands of data type byte, word, longword, quadword, octaword the operand is the zero extension of the 5-bit literal field:

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

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

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

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

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

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

where bits 63:32 are not present in a F_floating operand.
The exp and fra fields are used to form a G_floating operand as follows:

```
+-------+-------+-------+-------+-------+-------+
<table>
<thead>
<tr>
<th>1 1</th>
<th>5 4</th>
<th>4 3  1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024 + exp</td>
<td>fra 10</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:A+2</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:A+4</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:A+6</td>
</tr>
</tbody>
</table>
```

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

```
+-------+-------+-------+-------+-------+-------+
<table>
<thead>
<tr>
<th>1 1</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384 + exp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>fra</td>
<td>0</td>
<td>:A+2</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:A+4</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:A+6</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:A+8</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:A+10</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:A+12</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>:A+14</td>
</tr>
</tbody>
</table>
```

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

<table>
<thead>
<tr>
<th>E</th>
<th>F</th>
<th>-&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1/2</td>
<td>9/16</td>
</tr>
<tr>
<td>1</td>
<td>1/8</td>
<td>1 1/4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2 1/4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4 1/2</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>80</td>
</tr>
</tbody>
</table>

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 for literal mode is $S^\#\text{literal}$.

3.4.9 Index Mode

The operand specifier format is:

```
1 5 8 7 4 3 0
+-----------------+--------+
|                 | 4      | Rx     |
+-----------------+--------+
```

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

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

```
OA = BOA + \{\text{size} \times (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 calculation 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 Rx. PC cannot be used as an index register. If it is, a reserved addressing mode fault occurs (see Chapter 5). If the base operand specifier is for an addressing mode which results in register modification (i.e. autoincrement mode, autodecrement mode, or autoincrement deferred mode), the same register cannot be the index register. If it is, the primary operand address is UNPREDICTABLE.

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

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

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

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

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

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

6. byte, word, or longword displacement deferred indexed - @B^D(Rn)[Rx], @W^D(Rn)[Rx], or @L^D(Rn)[Rx]
### 3.5 SUMMARY OF GENERAL MODE ADDRESSING

#### 3.5.1 General Register Addressing

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Name</th>
<th>Assembler</th>
<th>rm</th>
<th>mw</th>
<th>av</th>
<th>PC</th>
<th>SP</th>
<th>AP&amp;FP</th>
<th>Indexable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>0-3</td>
<td>literal</td>
<td>S^#literal</td>
<td>y</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>indexed</td>
<td>i[Rx]</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>f</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>register</td>
<td>Rn</td>
<td>y</td>
<td>y</td>
<td>f</td>
<td>y</td>
<td>u</td>
<td>uq</td>
<td>uo</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>register deferred</td>
<td>(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>u</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>autodecrement</td>
<td>-(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>u</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>autoincrement</td>
<td>(Rn)+</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>deferred</td>
<td>$(Rn)+</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>byte displacement</td>
<td>B^D(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>byte displacement</td>
<td>$(B)^D(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>word displacement</td>
<td>W^D(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>word displacement</td>
<td>$(W)^D(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>longword displacement</td>
<td>L^D(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>longword displacement</td>
<td>$(L)^D(Rn)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>p</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>
### 3.5.2 Program Counter Addressing (reg=15)

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

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Name</th>
<th>Assembler</th>
<th>r</th>
<th>m</th>
<th>w</th>
<th>a</th>
<th>v</th>
<th>PC</th>
<th>SP</th>
<th>Indexable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>immediate</td>
<td>I^#constant</td>
<td>y</td>
<td>u</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>absolute</td>
<td>@#address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>byte relative</td>
<td>B^address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>byte relative</td>
<td>@B^address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>word relative</td>
<td>W^address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>word relative</td>
<td>@W^address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>long word relative</td>
<td>L^address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>long word relative</td>
<td>@L^address</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>

**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
- 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. +-----------------+
   | displ | 0 |
   +-----------------+
```

The operand specifier is a signed byte displacement.

```
2. +-----------------+
   | displ | 0 |
   +-----------------+
```

The operand specifier is a signed word displacement.

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

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

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

The following 3 steps are performed by each instruction:

1. Each operand specifier in order of instruction stream occurrence is treated as follows:
   a. If read access type: evaluate the operand address, read the operand, and save it.
   b. If write access type: evaluate the operand address and save it.
   c. If modify access type: evaluate the operand address and save it; read the operand and save it.
   d. If address access type: evaluate the address and save it.
   e. If branch access type: save the operand specifier.

2. Perform the operation indicated by the instruction.

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

NOTE

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

The implications of these conventions are:

1. Autoincrement and autodecrement operations occur as the operand specifiers are processed, and subsequent operand specifiers use the updated contents of registers modified by those operations.
2. Other than as indicated by 1, all input operands are read, and all addresses of output operands computed before any results of the instruction are stored.

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

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

12-Feb-82 -- Rev 7

4.1 INSTRUCTION SET

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

4.1.1 Instruction Descriptions

The instruction set is divided into 12 major sections:

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

11. Decimal string

12. Edit

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

1. The group name.

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

3. The operation of the instruction.

4. The effect on condition codes.

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

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

7. A description in English of the instruction.

8. Optional notes on the instruction and programming examples.
4.1.2 Operand Specifier Notation

Operand specifiers are described in the following way:

\[ <\text{name}>,<\text{access type}>,<\text{data type}> \]

where:

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

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

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

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

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

   r - Operand is read only.

   v - Calculate the effective address of the specified operand. If the effective address is in memory, the address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>. If the effective address is \( R_n \), the operand is in \( R_n \) or \( R_{n+1}'R_n \).

   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

w - word

x - first data type specified by instruction

y - second data type specified by instruction

4.1.3 Operation Description Notation

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

+  - addition

-  - subtraction, unary minus

*  - multiplication

/  - division (quotient only)

** - exponentiation

'  - concatenation

\<- - is replaced by

=  - is defined as

Rn or R[n] - contents of register Rn

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

PSW - the contents of the processor status word

PSL - the contents of the processor status long word

(x) - contents of memory location whose address is x
(x) - contents of memory location whose address is x;
    x incremented by the size of operand referenced
    at x
-(x) - x decremented by size of operand to be referenced
    at x; contents of memory location whose address is x
<x:y> - a modifier which delimits an extent from bit
    position x to bit position y inclusive
<x1,x2,...,xn> - a modifier which enumerates bits x1,x2,...,xn
{ } - arithmetic parentheses used to indicate precedence
AND - logical AND
OR - logical OR
XOR - logical XOR
NOT - logical (ones) complement
LSS - less than signed
LSSU - less than unsigned
LEQ - less than or equal signed
LEQU - less than or equal unsigned
EQL - equal signed
EQLU - equal unsigned
NEQ - not equal signed
NEQU - not equal unsigned
GEQ - greater than or equal signed
GEQU - greater than or equal unsigned
GTR - greater than signed
GTRU - greater than unsigned
SEXT(x) - x is sign extended to size of operand
    needed
ZEXT(x) - x is zero extended to size of operand needed
REM(x,y) - remainder of x divided by y, such that x/y and
    REM(x,y) have the same sign
MINU(x,y) - minimum unsigned of x and y
MAXU(x,y) - maximum unsigned of x and y

The following conventions are used:

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

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

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

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

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

The following instructions are described in this section.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Add Aligned Word</td>
<td>1</td>
</tr>
<tr>
<td>ADAWI add.rw, sum.mw</td>
<td></td>
</tr>
<tr>
<td>2. Add 2 Operands</td>
<td>3</td>
</tr>
<tr>
<td>ADD[B,W,L]2 add.rx, sum.mx</td>
<td></td>
</tr>
<tr>
<td>3. Add 3 Operands</td>
<td>3</td>
</tr>
<tr>
<td>ADD[B,W,L]3 add1.rx, add2.rx, sum.wx</td>
<td></td>
</tr>
<tr>
<td>4. Add With Carry</td>
<td>1</td>
</tr>
<tr>
<td>ADWC add.r1, sum.ml</td>
<td></td>
</tr>
<tr>
<td>5. Arithmetic Shift</td>
<td>2</td>
</tr>
<tr>
<td>ASH[L,Q] cnt.rb, src.rx, dst.wx</td>
<td></td>
</tr>
<tr>
<td>6. Bit Clear 2 Operands</td>
<td>3</td>
</tr>
<tr>
<td>BIC[B,W,L]2 mask.rx, dst.mx</td>
<td></td>
</tr>
<tr>
<td>7. Bit Clear 3 Operands</td>
<td>3</td>
</tr>
<tr>
<td>BIC[B,W,L]3 mask.rx, src.rx, dst.wx</td>
<td></td>
</tr>
<tr>
<td>8. Bit Set 2 Operands</td>
<td>3</td>
</tr>
<tr>
<td>BIS[B,W,L]2 mask.rx, dst.mx</td>
<td></td>
</tr>
<tr>
<td>9. Bit Set 3 Operands</td>
<td>3</td>
</tr>
<tr>
<td>BIS[B,W,L]3 mask.rx, src.rx, dst.wx</td>
<td></td>
</tr>
<tr>
<td>10. Bit Test</td>
<td>3</td>
</tr>
<tr>
<td>BIT[B,W,L] mask.rx, src.rx</td>
<td></td>
</tr>
<tr>
<td>11. Clear</td>
<td>4</td>
</tr>
<tr>
<td>CLR[B,W,L,Q] dst.wx</td>
<td></td>
</tr>
<tr>
<td>12. Compare</td>
<td>3</td>
</tr>
<tr>
<td>CMP[B,W,L] src1.rx, src2.rx</td>
<td></td>
</tr>
<tr>
<td>13. Convert</td>
<td>6</td>
</tr>
<tr>
<td>All pairs except BB,WW,LL.</td>
<td></td>
</tr>
<tr>
<td>14. Decrement</td>
<td>3</td>
</tr>
<tr>
<td>DEC[B,W,L] dif.mx</td>
<td></td>
</tr>
<tr>
<td>15. Divide 2 Operands</td>
<td>3</td>
</tr>
<tr>
<td>DIV[B,W,L]2 divr.rx, quo.mx</td>
<td></td>
</tr>
</tbody>
</table>
16. Divide 3 Operand  
\[ \text{DIV\{B,W,L\}3 divr.rx, divd.rx, quo.wx} \]  
3

17. Extended Divide  
\[ \text{EDIV divr.rl, divd.rq, quo.wl, rem.wl} \]  
1

18. Extended Multiply  
\[ \text{EMUL mulr.rl, muld.rl, add.rl, prod.wx} \]  
1

19. Increment  
\[ \text{INC\{B,W,L\} sum.mx} \]  
3

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

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

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

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

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

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

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

27. Rotate Long  
\[ \text{ROTL cnt.rb, src.rl, dst.wl} \]  
1

28. Add Aligned Word  
\[ \]  
1

29. Subtract With Carry  
\[ \text{SBWC sub.rl, dif.ml} \]  
1

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

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

32. Test  
\[ \text{TST\{B,W,L\} src.rx} \]  
3

33. Exclusive OR 2 Operand  
\[ \text{XOR\{B,W,L\}2 mask.rx, dst.mx} \]  
3
34. Exclusive OR 3 Operand
XOR{B,W,L}3 mask.rx, src.rx, dst.wx
ADAWI  Add Aligned Word Interlocked

Format:

opcode add.rw, sum.mw

Operation:

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

Condition Codes:

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

Exceptions:

reserved operand fault
integer overflow

Opcodes:

58   ADAWI  Add Aligned Word Interlocked

Description:

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

Notes:

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

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

Format:

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

Operation:

    sum <- sum + add;                        12 operand
    sum <- add1 + add2;                      13 operand

Condition Codes:

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

Exceptions:

    integer overflow

Opcodes:

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

Description:

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

Notes:

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

Format:

    opcode add.rl, sum.ml

Operation:

    sum ← sum + add + C;

Condition Codes:

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

Exceptions:

    integer overflow

Opcodes:

    D8   ADWC Add With Carry

Description:

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

Notes:

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

2. The 2 additions in the operation are performed simultaneously.
ASH Arithmetic Shift

Format:

\[
\text{opcode cnt.rb, src.rx, dst.wx}
\]

Operation:

\[
dst \leftarrow \text{src shifted cnt bits};
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow \text{dst LSS } 0; \\
Z & \leftarrow \text{dst EQL } 0; \\
V & \leftarrow \{\text{integer overflow}; \\
C & \leftarrow 0;
\end{align*}
\]

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 significant (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.
BIC    Bit Clear

Format:

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

Operation:

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

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

Condition Codes:

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

Exceptions:

None

Opcodes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8A</td>
<td>BICB2</td>
<td>Bit Clear Byte</td>
</tr>
<tr>
<td>8B</td>
<td>BICB3</td>
<td>Bit Clear Byte</td>
</tr>
<tr>
<td>AA</td>
<td>BICW2</td>
<td>Bit Clear Word</td>
</tr>
<tr>
<td>AB</td>
<td>BICW3</td>
<td>Bit Clear Word</td>
</tr>
<tr>
<td>CA</td>
<td>BICL2</td>
<td>Bit Clear Long</td>
</tr>
<tr>
<td>CB</td>
<td>BICL3</td>
<td>Bit Clear Long</td>
</tr>
</tbody>
</table>

Description:

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

Format:

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

Operation:

- dst <- dst OR mask; 12 operand
- dst <- src OR mask; 13 operand

Condition 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.
BIT Bit Test

Format:

opcode mask.rx, src.rx

Operation:

tmp <- src AND mask;

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

Format:

    opcode dst.wx

Operation:

    dst <- 0;

Condition Codes:

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

Exceptions:

    none

Opcodes:

    94  CLR B  Clear Byte
    B4  CLRW Clear Word
    D4  CLR L  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.
CMP  Compare

Format:

opcode src1.rx, src2.rx

Operation:

src1 - src2;

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

Format:

    opcode src.rx, dst.wy

Operation:

    dst <- conversion of src;

Condition Codes:

    N <- dst LSS 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.
DEC  Decrement

Format:

opcode dif.mx

Operation:

dif <- dif - 1;

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

2. DECx dif is equivalent to SUBx S^#1, dif, but is 1 byte shorter.
DIV     Divide

Format:

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

Operation:

quo <- quo / divr;              12 operand
quo <- divd / divr;              13 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:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>86</td>
<td>DIVB2 DiviDe Byte 2 Operands</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>DIVB3 DiviDe Byte 3 Operands</td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>DIVW2 DiviDe Word 2 Operands</td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td>DIVW3 DiviDe Word 3 Operands</td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>DIVL2 DiviDe Long 2 Operands</td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td>DIVL3 DiviDe Long 3 Operands</td>
<td></td>
</tr>
</tbody>
</table>

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

Exceptions:

terger 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.
EMUL  Extended Multiply

Format:

   opcode mulr.r1, muld.r1, add.r1, 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.
INC     Increment

Format:

opcode sum.mx

Operation:

sum <- sum + 1;

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

2. INCx sum is equivalent to ADDx S#1, sum, but is 1 byte shorter.
MCOM        Move Complemented

Format:

   opcode src.rx, dst.wx

Operation:

   dst <- NOT src;

Condition Codes:

   N <- dst LSS 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.
MNEG  Move Negated

Format:

    opcode src.rx, dst.wx

Operation:

    dst <- -src;

Condition Codes:

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

Exceptions:

    integer overflow

Opcodes:

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

Description:

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

Notes:

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

Format:

    opcode src.rx, dst.wx

Operation:

    dst <- src;

Condition Codes:

    N <- dst LSS 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    MOVQ  Move Octa

Description:

    The destination operand is replaced by the source operand.
MOVZ  Move Zero-Extended

Format:

opcode src.rx, dst.wy

Operation:

dst <- ZEXT(src);

Condition Codes:

N <- 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.
MUL     Multiply

Format:

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

Operation:

  prod <- prod * mulr;
        12 operand
  prod <- muld * mulr;
        13 operand

Condition Codes:

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

Exceptions:

  integer overflow

Opcodes:

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

Description:

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

Notes:

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

Format:

    opcode src.rl

Operation:

    -(SP) <- src;

Condition Codes:

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

Exceptions:

    none

Opcodes:

    DD   PUSHL Push Long

Description:

The longword source operand is pushed on the stack.

Notes:

PUSHL is equivalent to MOVL src, -(SP), but is 1 byte shorter.
ROTLP  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  ROTLP 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.
SBWC  Subtract With Carry

Format:

opcode sub.r1, 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.
SUB     Subtract

Format:

\[ \text{opcode } \text{sub.rx, dif.mx} \quad 2 \text{ operand} \]

\[ \text{opcode } \text{sub.rx, min.rx, dif.wx} \quad 3 \text{ operand} \]

Operation:

\[ \text{dif } \leftarrow \text{dif } - \text{sub}; \quad 12 \text{ operand} \]

\[ \text{dif } \leftarrow \text{min } - \text{sub}; \quad 13 \text{ operand} \]

Condition Codes:

\[ \text{N } \leftarrow \text{dif } \text{LSS } 0; \]
\[ \text{Z } \leftarrow \text{dif } \text{EQL } 0; \]
\[ \text{V } \leftarrow \text{integer overflow}; \]
\[ \text{C } \leftarrow \text{borrow into most significant bit}; \]

Exceptions:

integer overflow

Opnodes:

\[ \begin{array}{lll}
82 & \text{SUBB2} & \text{Subtract Byte 2 Operand} \\
83 & \text{SUBB3} & \text{Subtract Byte 3 Operand} \\
A2 & \text{SUBW2} & \text{Subtract Word 2 Operand} \\
A3 & \text{SUBW3} & \text{Subtract Word 3 Operand} \\
C2 & \text{SUBL2} & \text{Subtract Long 2 Operand} \\
C3 & \text{SUBL3} & \text{Subtract Long 3 Operand} \\
\end{array} \]

Description:

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

Notes:

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

Format:

    opcode src.rx

Operation:

    src - 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.
XOR      Exclusive OR

Format:

\text{opcode mask}.rx, \text{dst.mx} \quad 2 \text{ operand}

\text{opcode mask}.rx, \text{src.rx}, \text{dst.wx} \quad 3 \text{ operand}

Operation:

\text{dst} \leftarrow \text{dst XOR mask}; \quad \text{12 operand}

\text{dst} \leftarrow \text{src XOR mask}; \quad \text{13 operand}

Condition Codes:

\text{N} \leftarrow \text{dst LSS 0};
\text{Z} \leftarrow \text{dst EQL 0};
\text{V} \leftarrow 0;
\text{C} \leftarrow 0;

Exceptions:

none

Opcodes:

\begin{verbatim}
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
\end{verbatim}

Description:

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

The following instructions are described in this section.

1. Move Address
   \[\text{MOVA}\{B,W,L=F,Q=D=G,O=H\} \text{ src.ax, dst.wl}\]

2. Push Address
   \[\text{PUSHA}\{B,W,L=F,Q=D=G,O=H\} \text{ src.ax, } \{(SP).w1\}\]
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
  0E  MOVAL, Move Address Long
  8E  MOVAP  Move Address F_floating
  7E  MOVAQ, Move Address Quad
      MOVAD, Move Address D_floating
      MOVAG Move Address G_floating
  7EPD 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.
PUSHAX Push Address

Format:

opcode src.ax

Operation:

-(SP) <- src;

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

Notes:

1. PUSHAX src is equivalent to MOVAX src, -(SP), but is 1 byte shorter.

2. The source operand is of address access type which causes the address of the specified operand to be pushed.
4.4 VARIABLE LENGTH BIT FIELD INSTRUCTIONS

A variable length bit field is specified by 3 operands:

1. A longword position operand.

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

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

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

\[
\text{FIELD}(\text{pos}, \text{size}, \text{address}) = \begin{cases} 
\text{address} + \text{SEXT}(\text{pos}<31:3>) & \text{if address not specified by register mode} \\
R[n+1]'Rn<\{\text{size} - 1\} + \text{pos}:\text{pos} > & \text{if address specified by register mode and pos + size} \\
\end{cases}
\]

\(!\text{GTRU 32}
\]

\(!\text{LEQU 32}
\]

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

\[
1 + \left\{ \left\{ \text{size} - 1\right\} + \text{pos}<2:0> \right\} / 8 \right\}
\]

Zero bytes are referenced if the field size is 0.
The following instructions are described in this section.

1. Compare Field
   CMPV pos.r1, size.rb, base.vb, {field.rv}, src.r1

2. Compare Zero-Extended Field
   CMPZV pos.r1, size.rb, base.vb, {field.rv}, src.r1

3. Extract Field
   EXTV pos.r1, size.rb, base.vb, {field.rv}, dst.wl

4. Extract Zero-Extended Field
   EXTZV pos.r1, size.rb, base.vb, {field.rv}, dst.wl

5. Find First
   FF[S,C] startpos.r1, size.rb, base.vb, {field.rv}, findpos.wl

6. Insert Field
   INSV src.r1, pos.r1, size.rb, base.vb, {field.wv}

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

1. Branch on Bit
   BB[S,C] pos.r1, base.vb, displ.bb, {field.rv}

2. Branch on Bit (and modify without interlock)
   BB[S,C]{S,C} pos.r1, base.vb, displ.bb, {field.mv}

3. Branch on Bit (and modify) Interlocked
   BB[SS,CC]I pos.r1, base.vb, displ.bb, {field.mv}
CMP     Compare Field

Format:

opcode pos.rl, size.rb, base.vb, 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.
EXT Extract Field

Format:

\[ \text{opcode pos.rl, size.rb, base.vb, dst.wl} \]

Operation:

\[
\begin{align*}
\text{dst} & \leftarrow \text{if size NEQU } 0 \text{ then SEXT(FIELD(pos, size, base))} \\
& \quad \text{else } 0; \quad !\text{EXTV} \\
\text{dst} & \leftarrow \text{if size NEQU } 0 \text{ then ZEXT(FIELD(pos, size, base))} \\
& \quad \text{else } 0; \quad !\text{EXTZV}
\end{align*}
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow \text{dst LSS } 0; \\
Z & \leftarrow \text{dst EQL } 0; \\
V & \leftarrow 0; \\
C & \leftarrow C;
\end{align*}
\]

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.
FF    Find First

Format:

opcode startpos.rl, size.rb, base.vb, 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, size NEQ 0, and the field is contained in the registers.

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

Format:

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

Operation:

if size NEQU 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.
4.5 CONTROL INSTRUCTIONS

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

The following instructions are described in this section.

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

2. **Add One and Branch Less Than or Equal**
   \[ AOBLEQ \text{ limit.rl, index.ml, displ.bb} \]

3. **Add One and Branch Less Than**
   \[ AOBLS \text{ limit.rl, index.ml, displ.bb} \]

4. **Conditional Branch**
   \[ B\{\text{condition}\} \text{ displ.bb} \]

<table>
<thead>
<tr>
<th>Condition</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSS</td>
<td>Less Than</td>
</tr>
<tr>
<td>LEQ</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>EQL, EQLU</td>
<td>Equal, Equal Unsigned</td>
</tr>
<tr>
<td>NEQ, NEQU</td>
<td>Not Equal, Not Equal Unsigned</td>
</tr>
<tr>
<td>G&amp;Q</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>GTR</td>
<td>Greater Than</td>
</tr>
<tr>
<td>LSSU, CS</td>
<td>Less Than Unsigned, Carry Set</td>
</tr>
<tr>
<td>LEQU</td>
<td>Less Than or Equal Unsigned</td>
</tr>
<tr>
<td>GEQU, CC</td>
<td>Greater Than or Equal Unsigned, Carry Clear</td>
</tr>
<tr>
<td>GTRU</td>
<td>Greater Than Unsigned</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow Set</td>
</tr>
<tr>
<td>VC</td>
<td>Overflow Clear</td>
</tr>
</tbody>
</table>

5. **Branch on Bit**
   \[ BB\{S,C\} \text{ pos.rl, base.vb, displ.bb, \{field.rv\}} \]

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

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

8. **Branch on Low Bit**
   \[ BLB\{S,C\} \text{ src.rl, displ.bb} \]
9. Branch With {Byte, Word} Displacement
   BR{B,W} displ.bx

10. Branch to Subroutine With {Byte, Word} Displacement
    BSB{B,W} displ.bx, {-(SP).w1}

11. Case
    CASE{B,W,L} selector.rx, base.rx, limit.rx, displ.bw-list

12. Jump
    JMP dst.ab

13. Jump to Subroutine
    JSB dst.ab, {-(SP).w1}

14. Return from Subroutine
    RSB {(SP)+.rl}

15. Subtract One and Branch Greater Than or Equal
    SOBGEQ index.ml, displ.bb

16. Subtract One and Branch Greater Than
    SOBGTR index.ml, displ.bb
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

Opnodes:

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

Description:

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

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

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

3. On floating underflow, if FU is clear, the index operand is replaced by 0 and comparison and branch determination proceed normally. A fault occurs if FU is set and the index operand is unaffected.

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

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

6. Except for 5. above, the C-bit is unaffected.
AUBLEQ  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  AUBLEQ  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.11, index.m1, displ.bb

Operation:

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

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

F2  AOBLSS  Add One and Branch Less Than

Description:

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

Notes:

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

2. The C-bit is unaffected.
B     Branch on (condition)

Format:

    opcode displ.bb

Operation:

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

Condition Codes:

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

Exceptions:

    none

Opcodes:  Condition

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>{N OR Z} EQL 0</td>
<td>BCTR Branch on Greater Than (signed)</td>
</tr>
<tr>
<td>15</td>
<td>{N OR Z} EQL 1</td>
<td>BLEQ Branch on Less Than or Equal (signed)</td>
</tr>
<tr>
<td>12</td>
<td>Z EQL 0</td>
<td>BNEQ Branch on Not Equal (signed)</td>
</tr>
<tr>
<td>13</td>
<td>Z EQL 1</td>
<td>BNEQU Branch on Not Equal Unsigned</td>
</tr>
<tr>
<td>18</td>
<td>N EQL 0</td>
<td>BEQL Branch on Equal (signed)</td>
</tr>
<tr>
<td>19</td>
<td>N EQL 1</td>
<td>BEQLU Branch on Equal Unsigned</td>
</tr>
<tr>
<td>1A</td>
<td>{C OR Z} EQL 0</td>
<td>BGEQ Branch on Greater Than or Equal (signed)</td>
</tr>
<tr>
<td>1B</td>
<td>{C OR Z} EQL 1</td>
<td>BLSS Branch on Less Than (signed)</td>
</tr>
<tr>
<td>1C</td>
<td>V EQL 0</td>
<td>BGTRU Branch on Greater Than Unsigned</td>
</tr>
<tr>
<td>1D</td>
<td>V EQL 1</td>
<td>BLEQ Branch Less Than or Equal Unsigned</td>
</tr>
<tr>
<td>1E</td>
<td>C EQL 0</td>
<td>BVC Branch on Overflow Clear</td>
</tr>
<tr>
<td>1F</td>
<td>C EQL 1</td>
<td>BVS Branch on Overflow Set</td>
</tr>
</tbody>
</table>

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
   BGTU   {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.
BB  Branch on Bit

Format:

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

Operation:

    teststate = if {BBS} then 1 else 0;
    if FIELD(pos, l, 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.r1, base.vb, 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:

\[
\text{opcode pos.rl, base.vb, displ.bb}
\]

Operation:

\[
\begin{align*}
teststate &= \text{if } \{\text{BBSSI}\} \text{ then } 1 \text{ else } 0; \\
\text{newstate} &= \text{teststate}; \\
\{\text{set interlock}\}; \\
\text{tmp} &= \text{FIELD(pos, 1, base)}; \\
\text{FIELD(pos, 1, base)} &= \text{newstate}; \\
\{\text{release interlock}\}; \\
\text{if } \text{tmp} \text{ EQL teststate then} \\
\text{PC} &= \text{PC} + \text{SEXT(displ)};
\end{align*}
\]

Condition Codes:

\[
\begin{align*}
\text{N} &= \text{N}; \\
\text{Z} &= \text{Z}; \\
\text{V} &= \text{V}; \\
\text{C} &= \text{C};
\end{align*}
\]

Exceptions:

reserved operand

Opcodes:

\[
\begin{align*}
\text{E6} & \quad \text{BBSSI Branch on Bit Set and Set Interlocked} \\
\text{E7} & \quad \text{BBCCI Branch on Bit Clear and Clear Interlocked}
\end{align*}
\]

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:

\[
\text{teststate} = \begin{cases} 
1 & \text{if } \{\text{BLBS}\} \text{ then } \text{teststate} = 1 \text{ else } 0; \\
\text{if } src<0> \text{ EQL teststate then} \\
PC \leftarrow PC + \text{SEXT(disp1)};
\end{cases}
\]

Condition Codes:

- \(N \leftarrow N\)
- \(Z \leftarrow Z\)
- \(V \leftarrow V\)
- \(C \leftarrow C\)

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.
BR Branch

Format:
opcode displ.bx

Operation:
PC <- PC + SEXT(displ);

Condition Codes:
N <- N;
Z <- Z;
V <- V;
C <- C;

Exceptions:
none

Opcodes:

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

Description:
The sign-extended branch displacement is added to PC and PC is replaced by the result.
BSB Branch To Subroutine

Format:

    opcode displ.bx

Operation:

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

Condition Codes:

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

Exceptions:

    none

Opcodes:

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

Description:

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

\[
\text{opcode selector.rx, base.rx, limit.rx, displ[0].bw, ..., displ[limit].bw}
\]

Operation:

\[
\begin{align*}
tmp & \leftarrow \text{selector} - \text{base}; \\
\text{PC} & \leftarrow \text{PC} + \text{if tmp LEQU limit then} \\
& \qquad \text{SEXT(displ[tmp]) else } \{2 + 2 \ast \text{ZEXT(limit)}\};
\end{align*}
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow \text{tmp LSS limit}; \\
Z & \leftarrow \text{tmp EQL limit}; \\
V & \leftarrow \emptyset; \\
C & \leftarrow \text{tmp LSSU limit};
\end{align*}
\]

Exceptions:

none

Opcodes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8F</td>
<td>CASEB</td>
<td>Case Byte</td>
</tr>
<tr>
<td>AF</td>
<td>CASEW</td>
<td>Case Word</td>
</tr>
<tr>
<td>CF</td>
<td>CASEL</td>
<td>Case Long</td>
</tr>
</tbody>
</table>

Description:

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

Notes:

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

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

Format:
    opcode dst.ab

Operation:
    PC <- dst;

Condition Codes:
    N <- N;
    Z <- Z;
    V <- V;
    C <- C;

Exceptions:
    none

Opcodes:
    17   JMP      Jump

Description:
    PC is replaced by the destination operand.
JSB     Jump to Subroutine

Format:

    opcode dst.ab

Operation:

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

Condition Codes:

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

Exceptions:

    none

Opcodes:

    16   JSB     Jump to Subroutine

Description:

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

Notes:

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

Format:

   opcode

Operation:

   PC <- (SP)+;

Condition Codes:

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

Exceptions:

   none

Opcodes:

   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.
SUBGEQ  Subtract One and Branch Greater Than or Equal

Format:

opcode index.m1, 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  SUBGEQ  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:

\text{opcode} \ index.m1, \ displ.bb

Operation:

\text{index} \leftarrow \text{index} - 1;
\text{if index GTR 0 then } \text{PC} \leftarrow 
\text{PC + SEXT(displ)};

Condition Codes:

\begin{align*}
\text{N} & \leftarrow \text{index LSS 0}; \\
\text{Z} & \leftarrow \text{index EQL 0}; \\
\text{V} & \leftarrow \{\text{integer overflow}\}; \\
\text{C} & \leftarrow \text{C}; \\
\end{align*}

Exceptions:

integer overflow

Opcodes:

\begin{itemize}
  \item[F5] SOBGTR  Subtract One and Branch Greater Than
\end{itemize}

Description:

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

Notes:

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

2. The C-bit is unaffected.
4.6 PROCEDURE CALL INSTRUCTIONS

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

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

```
1 1 1 1 1
5 4 3 2 1
```

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. In addition, PC, SP, FP, and AP are always preserved by the CALL instructions and restored by the RET instruction.

All external procedure CALLS generated by standard DIGITAL language processors, and all inter-module CALLS to major VAX-11 software subsystems comply with the procedure calling software standard (see VAX/VMS Run Time Library Reference Manual, Appendix C). The procedure calling standard requires that all registers in the range R2 through R11 used in the procedure must appear in the mask. R0 and R1 are not preserved by any called procedure that complies with the procedure calling standard.

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

condition handler (initially 0)  :(FP)
<table>
<thead>
<tr>
<th>SPA</th>
<th>S</th>
<th>0</th>
</tr>
</thead>
</table>
mask<11:0> | saved PSW<15:5> | 0 |
|---|---|---|
saved AP
|---|
saved FP
|---|
saved PC
|---|
saved R0 (...)
|---|

0 to 3 bytes specified by SPA, Stack Pointer Alignment

S = set if CALLS; clear if CALLG.

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

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

The following instructions are described in this section.

1. Call Procedure with General Argument List
   CALLG arglist.ab, dst.ab, {-(SP).w*}

2. Call Procedure with Stack Argument List
   CALLS numarg.r1, dst.ab, {-(SP).w*}

3. Return from Procedure
   RET {(SP)+.r*}

Instructions
CALLG Call Procedure With General Argument List

Format:
opcode arglist.ab, dst.ab

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

Condition Codes:
N ← 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.
Instructions 12-Feb-82 -- Rev 7
PROCEDURE CALL INSTRUCTIONS

stack

frame

(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 VAX/VMS Run Time Library Reference Manual, Appendix C.
CALLS  Call Procedure with Stack Argument List

Format:

opcode numarg.rl, dst.ab

Operation:

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

Condition Codes:

N <- 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 VAX/VMS Run Time Library Reference Manual, Appendix C.
RET Return from Procedure

Format:

 opcode

Operation:

 [restore SP from FP];
 [restore registers];
 [drop stack alignment];
 [if CALLS then remove arglist];
 [restore PSW];

Condition Codes:

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

Exceptions:

 reserved operand

Opcodes:

 Ø4 RET Return from Procedure

Description:

SP is replaced by FP plus 4. A longword containing stack alignment bits in bits 31:30, a CALLS/CALLC 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, FP, and AP are replaced by longwords popped from the stack. A register restore mask is formed from bits 27:16 of the temporary. Scanning from bit 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 tmpl<15:8> NEQ 0.

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

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

4. The procedure calling standard and condition handling facility assume that procedures which return a function value or a status code do so in R0 or R0 and R1. Refer to VAX/VMS Run Time Library Reference Manual, Appendix C.
4.7 MISCELLANEOUS INSTRUCTIONS

The following instructions are described in this section.

1. Bit Clear PSW
   BICPSW mask.rw

2. Bit Set PSW
   BISPSW mask.rw

3. Breakpoint Fault
   BPT {-(KSP).w*}

4. Halt
   HALT {-(KSP).w*}

5. Index
   INDEX subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl

6. Move from PSL
   MOVPSL dst.wl

7. No Operation
   NOP

8. Pop Registers
   POPR mask.rw, {(SP)+.r*}

9. Push Registers
   PUSHR mask.rw, {-(SP).w*}

10. Extended Function Call
    XFC {unspecified operands}
BICPSW  Bit Clear PSW

Format:
  opcode mask rw

Operation:
  PSW ← PSW AND {NOT mask};

Condition Codes:
  N ← N AND {NOT mask<3>};
  Z ← Z AND {NOT mask<2>};
  V ← V AND {NOT mask<1>};
  C ← C AND {NOT mask<0>};

Exceptions:
  reserved operand

Opcodes:
  B9  BICPSW  Bit Clear PSW

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

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

Format:

opcode mask rw

Operation:

PSW <- PSW OR mask;

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

B8  BISPSW  Bit Set PSW

Description:

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

Notes:

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

Format:

    opcode

Operation:

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

Condition Codes:

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

Exceptions:

    none

Opcodes:

    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 debugging facilities.
HALT     Halt

Format:

opcode

Operation:

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

Condition Codes:

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

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

Exceptions:

privileged instruction

Opcodes:

00    HALT    Halt

Description:

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

Notes:

This opcode is 0 to trap many branches to data.
INDEX Compute Index

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

    subscript range

Opcodes:

    6A   INDEX   index

Description:

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

Notes:

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

2. The index instruction is useful in index calculations for arrays of the fixed length data types (integer and floating) and for index calculations for arrays of bit fields, character strings, and decimal strings. The indexin operand permits cascading INDEX instructions for multidimensional arrays. For
one-dimensional bit field arrays it also permits introduction of the constant portion of an index calculation which is not readily absorbed by address arithmetic. The following notes will show some of the uses of INDEX.

3. The COBOL statements:

```cobol
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:

```pl1
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:

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

could compile to:

```
INDEX J, #L2, #U2, #M1, #0, R0; M1=U1-L1+1
INDEX I, #L1, #U1, #1, R0, R0;
MOVL #1, A-a[R0]; a = [(L2*M1) + L1] *4
```
MOVPSL  Move from PSL

Format:
opcode dst.w1

Operation:
dst <- PSL;

Condition Codes:
N <- N;
Z <- Z;
V <- V;
C <- C;

Exceptions:
none

Opcodes:
DC     MOVPSL  Move from PSL

Description:
The destination operand is replaced by PSL (See Chapter 6).
NOP      No Operation

Format:

    opcode

Operation:

    none

Condition Codes:

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

Exceptions:

    none

Opcodes:

    01    NOP      No Operation

Description:

    No operation is performed.
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.
PUSHR    Push Registers

Format:

opcode mask.rw

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcodes:

BB    PUSHR    Push Registers

Description:

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

Notes:

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

Format:

opcode

Operation:

{XFC fault};

Condition Codes:

N ← 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.
4.8 QUEUE INSTRUCTIONS

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

4.8.1 Absolute Queues

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

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

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

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

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

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

```
| 3 |
+-------------------------+----------+       |
| B                       | :B       |
+-------------------------+----------+       |
| H                       | :H       |
```

If an entry at address C is inserted into the queue above:

```
| 3 |
+-------------------------+----------+       |
| C                       | :C       |
+-------------------------+----------+       |
| B                       | :B       |
| H                       | :H       |
```

If an entry at address D is inserted into the queue above:

```
| 3 |
+-------------------------+----------+       |
| D                       | :D       |
+-------------------------+----------+       |
| C                       | :C       |
| B                       | :B       |
| H                       | :H       |
```
If an entry at address A is inserted at the head of the queue, the queue is as shown below:

```
3 1
+-------------------+   0
|                   |
|                   |
| A                 |   :H
3 1
+-------------------+   0
|                   |
|                   |
| B                 |   :H+4
3 1
+-------------------+   0
|                   |
|                   |
|                   |
3 1
+-------------------+   0
|                   |
|                   |
| B                 |   :A
3 1
+-------------------+   0
|                   |
|                   |
| H                 |   :A+4
3 1
+-------------------+   0
|                   |
|                   |
|                   |
3 1
+-------------------+   0
|                   |
|                   |
| H                 |   :B
3 1
+-------------------+   0
|                   |
|                   |
| A                 |   :B+4
```
Finally, if an entry at address C is inserted at the tail, the queue appears as follows:

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

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

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

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

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

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

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

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

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

```
3
1
```

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

```
3
1
```

```
3
1
```

```
3
1
```

```
3
1
```
If an entry at address A is inserted at the head of the queue, the queue is as shown below:

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

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

```
3
1
+-------------------------+ 0
|                         |
|  H - B                  | :B
|                         |
|  A - B                  | :B+4
|                         |
3
1
```
Finally, if an entry at address C is inserted at the tail, the queue appears as follows:

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

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

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

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

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

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

4.8.3 Instruction Descriptions

The following instructions are described in this section.

<table>
<thead>
<tr>
<th>Instruction Description</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Insert Entry into Queue at Head, Interlocked</td>
<td>1</td>
</tr>
<tr>
<td>INSQHI entry.ab, header.aq</td>
<td></td>
</tr>
<tr>
<td>2. Insert Entry into Queue at Tail, Interlocked</td>
<td>1</td>
</tr>
<tr>
<td>INSQTI entry.ab, header.aq</td>
<td></td>
</tr>
<tr>
<td>3. Insert Entry in Queue</td>
<td>1</td>
</tr>
<tr>
<td>INSQUE entry.ab, pred.ab</td>
<td></td>
</tr>
<tr>
<td>4. Remove Entry from Queue at Head, Interlocked</td>
<td>1</td>
</tr>
<tr>
<td>REMQHI header.aq, addr.wl</td>
<td></td>
</tr>
<tr>
<td>5. Remove Entry from Queue at Tail, Interlocked</td>
<td>1</td>
</tr>
<tr>
<td>REMQTI header.aq, addr.wl</td>
<td></td>
</tr>
<tr>
<td>6. Remove Entry from Queue</td>
<td>1</td>
</tr>
<tr>
<td>REMQUE entry.ab, addr.wl</td>
<td></td>
</tr>
</tbody>
</table>
INSQHI  Insert Entry into Queue at Head, Interlocked

Format:

opcode  entry.ab, header.aq

Operation:

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

if tmpl<>0  EQLU 1 then

interlock

begin  (header){interlocked} <- tmpl;  !release hardware
{set condition codes and terminate instruction};
end;
else
interlock

begin  (header){interlocked} <- tmpl  v 1;  !set secondary
!release hardware
interlock

If {all memory accesses can be completed} then
!check if following addresses can be written
!without causing a memory management exception:
!  entry
!  header + tmpl
!Also, check for quadword alignment
begin
{insert entry into queue};
{release secondary interlock};
end;
else
begin
{release secondary interlock};
{backup instruction};
{initiate fault};
end;
end;
Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

5C    INSQHI Insert Entry into Queue at Head, Interlocked

Description:

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

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

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

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

\[
\text{INSERT:} \quad \text{INSQHI } \ldots \quad ;\text{was queue empty?} \\
\quad \text{BEQL} \ 1\$ \quad ;\text{yes} \\
\quad \text{BCS} \ \text{INSERT} \quad ;\text{try inserting again} \\
\quad \text{CALL} \ \text{WAIT(\ldots)} \quad ;\text{no, wait}
\]

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 \((\text{header})<2:1>\) is not zero. A reserved operand fault also occurs if header equals entry. In this case the queue is not altered.
INSQTI  Insert Entry into Queue at Tail, Interlocked

Format:

opcode   entry.ab, header.aq

Operation:

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

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

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

\[
\text{if \{insertion succeeded\} then}
\]

\[
\text{begin}
N \leftarrow 0;
Z \leftarrow (\text{entry}) \text{ EQL (entry+4)}; \quad \text{!first entry in queue}
V \leftarrow 0;
C \leftarrow 0;
\text{end;}
\]

\[
\text{else}
\]

\[
\text{begin}
N \leftarrow 0;
Z \leftarrow 0;
V \leftarrow 0;
C \leftarrow 1; \quad \text{!secondary interlock failed}
\text{end;}
\]

Exceptions:

reserved operand

Opcodes:

5D \quad \text{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.
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  l$ ;yes
      BCS INSERT ;try inserting again
      CALL   WAIT(...) ;no, wait

   l$:

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.
INSQUE: Insert Entry in Queue

Format:

opcode  entry.ab, pred.ab

Operation:

If (all memory accesses can be completed) then
begin
(entry) <- (pred);
(entry + 4) <- pred;
((pred) + 4) <- entry;
(pred) <- entry;
end;
else
begin
{backup instruction};
{initiate fault};
end;

Condition Codes:

N <- (entry) LSS (entry+4);  !first entry in queue
Z <- (entry) EQL (entry+4);   
V <- 0;
C <- (entry) LSSU (entry+4);

Exceptions:

none

Opcodes:

0E       INSQUE: Insert Entry in Queue

Description:

The entry specified by the entry operand is inserted into the queue following the entry specified by the predecessor operand. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a 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 l$ ;yes
   CALL WAIT(...) ;no, wait

   l$:

5. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion is not started.
REMQHI  Remove Entry from Queue at Head, Interlocked

Format:

\[
\text{opcode header.aq, addr.wl}
\]

Operation:

\[
\text{tmpl <- (header)(interlocked);}
\]

\[
\text{!acquire hardware interlock}
\]

\[
\text{!must have write access to header}
\]

\[
\text{!header must be quadword aligned}
\]

\[
\text{!header cannot equal address of tmpl<2:1> must be zero}
\]

\[
\text{if tmpl<Ø> ØLQV 1 then begin}
\]

\[
\text{(header)(interlocked) <- tmpl;}
\]

\[
\text{!release hardware interlock}
\]

\[
\text{!set condition codes and terminate instruction};
\]

\[
\text{end;}
\]

\[
\text{else begin}
\]

\[
\text{(header)(interlocked) <- tmpl v 1;}
\]

\[
\text{!set secondary interlock}
\]

\[
\text{!release hardware interlock}
\]

\[
\text{If [all memory accesses can be completed] then begin}
\]

\[
\text{!check if the following can be done without causing a memory management exception:}
\]

\[
\text{!write addr operand}
\]

\[
\text{!read contents of header + tmpl \{if tmpl NEQ 0\}}
\]

\[
\text{!write into header + tmpl + (header + tmpl) \{if \}
\]

\[
\text{!tmpl NEQ 0\}}
\]

\[
\text{!Also, check for quadword alignment begin}
\]

\[
\text{!remove entry from queue};
\]

\[
\text{!release secondary interlock};
\]

\[
\text{end;}
\]

\[
\text{else begin}
\]

\[
\text{!release secondary interlock};
\]

\[
\text{!backup instruction};
\]

\[
\text{!initiate fault};
\]

\[
\text{end;}
\]
Condition Codes:

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

Exceptions:

    reserved operand

Opcodes:

5E  REMQHI  Remove Entry from Queue at Head, Interlocked

Description:

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

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

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

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

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

   2$:

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

   1$:  REMQHI ... ; anything removed?
   BVS   2$ ; no
   .
   process removed entry
   .
   BR    1$ ;
   .

   2$:  BCS   1$ ; try removing again
   queue empty

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

6. A reserved operand fault occurs if header or (header + (header)) is an address that is not quadword aligned (i.e. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if the header address operand equals the address of the addr operand. In this case the queue is not altered.
REMQTI  Remove Entry from Queue at Tail, Interlocked

Format:

    opcode  header.aq, addr.w1

Operation:

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

    if tmpl<0> EQLU 1 then
        begin
            (header){interlocked} <= tmpl;  !release hardware
            interlock
                {set condition codes and terminate instruction};
            end;
        else
            begin
                (header){interlocked} <= tmpl v 1;  !set secondary
                interlock                     !release hardware
            end;
    end;

    If {all memory accesses can be completed} then
    !check if the following can be done without
    !causing a memory management exception:
    !write addr operand
    !read contents of header + (header + 4) {if tmpl
    !                   NEQU 0}
    !write into header + (header + 4)
    ![header + 4 + (header + 4)) {iftmpl NEQU
    0}
    !Also, check for quadword alignment
    begin
        [remove entry from queue];
        [release secondary interlock];
    end;
    else
        begin
            [release secondary interlock];
            [backup instruction];
            [initiate fault];
        end;
    end;
Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

5F    REMQTI    Remove Entry from Queue at Tail, Interlocked

Description:

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

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

2. The INSOHI, 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 the header address
   operand equals the address of the addr operand. In this case
   the queue is not altered.
REMQUE Remove Entry From Queue

Format:

opcode entry.ab,addr.wl

Operation:

if {all memory accesses 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);
C <- (entry) LSSU (entry+4);  !no entry to remove

Exceptions:

none

Opcodes:

ØF REMQUE Remove Entry from Queue

Description:

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

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

   1. Remove at head
      
      REMQUE @h,addr
      ; h is queue header

   2. Remove at tail
      
      REMQUE @h+4,addr
      ; h is queue header

   3. Remove arbitrary entry
      
      REMQUE entry,addr

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

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

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

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

1$:

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

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

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

The floating point instructions operate on four data types. F_floating and D_floating instructions are standard on all VAX processors. G_floating and H_floating instructions are optional on the VAX-11/780 and the VAX-11/750; standard on the VAX-11/730.

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., autodecrement, autodecrement, or autoincrement deferred), the value of the floating point operand is UNPREDICTABLE.

4.9.1 Introduction

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

\[(+ \text{ or } -)(2**K)*f,\]

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

\[1/2 \text{ LEQ } f \text{ LSS 1.}\]

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 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
quadrupe precision, or H floating, data is 128 bits long. Sign magnitude notation is used, as follows:

1. Non-zero floating point numbers:

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

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

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

In the G floating data type eleven bits are reserved for the storage of the exponent in excess 1024 notation. In the H floating data type fifteen bits are reserved for the storage of the exponent in excess 16384 notation. A biased exponent of 0 is reserved for floating point zero. Thus, exponents are restricted to -1023 to +1023 inclusive (in excess notation, 1 to 2047), and -16383 to +16383 inclusive (in excess notation, 1 to 32767) for the G floating and H floating data types respectively.

2. Floating point zero:

Because of the hidden bit, the fractional factor is not available to distinguish between zero and non-zero numbers whose fractional factor is exactly 1/2. Therefore the VAX-11 reserves a sign-exponent field of 0 for this purpose. Any positive floating point number with biased exponent of 0 is treated as if it were an exact 0 by the floating point instruction set. In particular, a floating point operand, whose bits are all 0's, is treated as zero, and this is the format generated by all floating point instructions for which the result is zero.

3. The Reserved Operands:

A reserved operand is defined to be any bit pattern with a sign bit of one and a biased exponent of zero. On the VAX-11, all floating point instructions generate a fault if a reserved operand is
encountered. A reserved operand is never generated as a result of a floating point instruction.

4.9.2 Overview Of The Instruction Set

The VAX-11 has the standard arithmetic operations ADD, SUB, MUL, and DIV implemented for all four floating data types. The results of these operations are always rounded, as described in the section on accuracy. It has, in addition, two composite operations, EMOD and POLY, also implemented for all four floating point data types. EMOD generates a product of two operands, and then separates the product into its integer and fractional terms. POLY evaluates a polynomial, given the degree, the argument and pointer to a table of coefficients. Details on the operation of EMOD and POLY are given in their respective descriptions. All of these instructions are subject to the rounding errors associated with floating point operations, as well as to exponent overflow and underflow. Accuracy is discussed in the next section, and exceptions are discussed in Chapter 6.

The VAX-11 also has a complete set of instructions for conversion from integer arithmetic types (byte, word, longword) to all floating types (F floating, D floating, G floating, H floating), and vice versa. The VAX-11 also has a set of Instructions for conversion between all of the floating types except between D floating and G floating. Many of these instructions are exact, in the sense defined in the section on accuracy to follow. However, a few may generate rounding error, floating overflow, floating underflow, or induce integer overflow. Details are given in the description of the CVT instructions.

There is a class of move-type instructions which are always exact: MOV, NEG, CLR, CMP, and TST. And, finally, there is the ACB (add, compare and branch) instruction, which is subject to rounding errors, overflow and underflow.

All of the floating point instructions on the VAX-11 fault if a reserved operand is encountered. Floating point instructions also fault on the occurrence of floating overflow or divide by zero, and the condition codes are UNPREDICTABLE. The FU bit, in the PSW, is available to enable or disable an exception on underflow. If the FU bit is clear, no exception occurs on underflow and zero is returned as the result. If the FU bit is set, a fault occurs on underflow. Further details on the actions taken if any of these exceptions occurs are included in the descriptions of the instructions, and completely discussed in Chapter 6.

4.9.3 Accuracy

General comments on the accuracy of the VAX-11 floating point instruction set are presented here. The descriptions of the individual instructions may include additional details on the accuracy at which they operate.
An instruction is defined to be exact if its result, extended on the right by an infinite sequence of zeroes, is identical to that of an infinite precision calculation involving the same operands. The a priori accuracy of the operands is thus ignored. For all arithmetic operations, except DIV, a zero operand implies that the instruction is exact. The same statement holds for DIV if the zero operand is the dividend. But if it is the divisor, division is undefined and the instruction faults.

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

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

1. If the rounding bit is one, the rounded result is the chopped result incremented by an LSB (least significant bit).

2. If the rounding bit is zero, the rounded and chopped results are identical.

All VAX-11 processors implement rounding so as to produce results identical to the results produced by the following algorithm. Add a 1 to the rounding bit, and propagate the carry, if it occurs. Note that a renormalization may be required after rounding takes place; if this happens, the new rounding bit will be zero, so it can happen only once. The following statements summarize the relations among chopped, rounded and true (infinite precision) results:

1. If a stored result is exact
   rounded value = chopped value = true value.

2. If a stored result is not exact, it's magnitude
   1. is always less than that of the true result for chopping.
2. is always less than that of the true result for rounding if the rounding bit is zero.

3. is greater than that of the true result for rounding if the rounding bit is one.
4.9.4 Instruction Descriptions

The following instructions are described in this section.

<table>
<thead>
<tr>
<th>Instruction Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Add 2 Operand</td>
<td>4</td>
</tr>
<tr>
<td>ADD{F,D,G,H}2 add.rx, sum.mx</td>
<td></td>
</tr>
<tr>
<td>2. Add 3 Operand</td>
<td>4</td>
</tr>
<tr>
<td>ADD{F,D,G,H}3 add1.rx, add2.rx, sum.wx</td>
<td></td>
</tr>
<tr>
<td>3. Clear</td>
<td>3</td>
</tr>
<tr>
<td>CLR{L=F,Q=D=G,O=H} dst.wx</td>
<td></td>
</tr>
<tr>
<td>4. Compare</td>
<td>4</td>
</tr>
<tr>
<td>CMP{F,D,G,H} src1.rx, src2.rx</td>
<td></td>
</tr>
<tr>
<td>5. Convert</td>
<td>34</td>
</tr>
<tr>
<td>CVT{F,D,G,H}{B,W,L,F,D,G,H} src.rx, dst.wy</td>
<td></td>
</tr>
<tr>
<td>CVT{B,W,L}{F,D,G,H} src.rx, dst.wy</td>
<td></td>
</tr>
<tr>
<td>All pairs except FF,DD,GG,HH,DG, and GD</td>
<td></td>
</tr>
<tr>
<td>6. Convert Rounded</td>
<td>4</td>
</tr>
<tr>
<td>CVTR{F,D,G,H}L src.rx, dst.wl</td>
<td></td>
</tr>
<tr>
<td>7. Divide 2 Operand</td>
<td>4</td>
</tr>
<tr>
<td>DIV{F,D,G,H}2 divr.rx, quo.mx</td>
<td></td>
</tr>
<tr>
<td>8. Divide 3 Operand</td>
<td>4</td>
</tr>
<tr>
<td>DIV{F,D,G,H}3 divr.rx, divd.rx, quo.wx</td>
<td></td>
</tr>
<tr>
<td>9. Extended Modulus</td>
<td>4</td>
</tr>
<tr>
<td>EMOD{F,D} mulr.rx, mulrx.rb, muld.rx, int.wx, fract.wx</td>
<td></td>
</tr>
<tr>
<td>EMOD{G,H} mulr.rx, mulrx.rb, muld.rx, int.wx, fract.wx</td>
<td></td>
</tr>
<tr>
<td>10. Move Negated</td>
<td>4</td>
</tr>
<tr>
<td>MNEG{F,D,G,H} src.rx, dst.wx</td>
<td></td>
</tr>
<tr>
<td>11. Move</td>
<td>4</td>
</tr>
<tr>
<td>MOV{F,D,G,H} src.rx, dst.wx</td>
<td></td>
</tr>
<tr>
<td>12. Multiply 2 Operand</td>
<td>4</td>
</tr>
<tr>
<td>MUL{F,D,G,H}2 mulr.rx, prod.mx</td>
<td></td>
</tr>
<tr>
<td>13. Multiply 3 Operand</td>
<td>4</td>
</tr>
<tr>
<td>MUL{F,D,G,H}3 mulr.rx, muld.rx, prod.wx</td>
<td></td>
</tr>
<tr>
<td>14. Polynomial Evaluation F_floating</td>
<td>1</td>
</tr>
<tr>
<td>POLYF arg.rf, degree.rw, tbladdr.ab, {R0-3.wl}</td>
<td></td>
</tr>
</tbody>
</table>
FLOATING POINT INSTRUCTIONS

15. Polynomial Evaluation D_floating
   POLYD arg.rd, degree.rw, tbladdr.ab, \{R0-5.wl\}

16. Polynomial Evaluation G_floating
   POLYG arg.rg, degree.rw, tbladdr.ab, \{R0-5.wl\}

17. Polynomial Evaluation H_floating
   POLYH arg.rh, degree.rw, tbladdr.ab,
   \{R0-5.wl, -16(SP):=1(SP).wb\}

18. Subtract 2 Operand
   SUB\{F,D,G,H\}2 sub.rx, dif.mx

19. Subtract 3 Operand
   SUB\{F,D,G,H\}3 sub.rx, min.rx, dif.wx

20. Test
    TST\{F,D,G,H\} src.rx

The following floating point instructions are described in the section on Control Instructions.

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

Format:

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

Operation:

sum <= sum + add;             !2 operand
sum <= add1 + add2;          !3 operand

Condition Codes:

N <= sum LSS 0;
Z <= sum EQL 0;
V <= {floating overflow};
C <= 0;

Exceptions:

floating overflow
floating underflow
reserved operand

OpCodes:

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

Description:

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

Notes:

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

2. On floating underflow, if FU is set a fault occurs. Zero is
   stored as the result of floating underflow only if FU is clear.
   On a floating underflow fault, the sum operand is unaffected.
   If FU is clear, the sum operand is replaced by 0 and no
exception occurs.

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

Format:

    opcode dst.wx

Operation:

    dst <- 0;

Condition Codes:

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

Exceptions:

    none

Opcodes:

    D4  CLR F_floating
    7C  CLR D_floating,
    CLRG Clear G_floating
    7CFD CLR 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.
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   CMFH    Compare H_floating

Description:

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

Notes:

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

Format:

opcode src.rx, dst.wy

Operation:

dst <- conversion of src;

Condition Codes:

N <- dst LSS 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     CVTBDG 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     CVTLLG Convert Long to G_floating
6EFD     CVTLLH Convert Long to H_floating
**FLOATING POINT INSTRUCTIONS**

<table>
<thead>
<tr>
<th>Code</th>
<th>Instruction</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>CVTFB</td>
<td>Convert FFloating to Byte</td>
</tr>
<tr>
<td>68</td>
<td>CVTDB</td>
<td>Convert DFloating to Byte</td>
</tr>
<tr>
<td>48FD</td>
<td>CVTGB</td>
<td>Convert GFloating to Byte</td>
</tr>
<tr>
<td>68FD</td>
<td>CVTHB</td>
<td>Convert HFloating to Byte</td>
</tr>
<tr>
<td>49</td>
<td>CVTFW</td>
<td>Convert FFloating to Word</td>
</tr>
<tr>
<td>69</td>
<td>CVTDW</td>
<td>Convert DFloating to Word</td>
</tr>
<tr>
<td>49FD</td>
<td>CVTGW</td>
<td>Convert GFloating to Word</td>
</tr>
<tr>
<td>69FD</td>
<td>CVTHW</td>
<td>Convert HFloating to Word</td>
</tr>
<tr>
<td>4A</td>
<td>CVTFL</td>
<td>Convert FFloating to Long</td>
</tr>
<tr>
<td>4B</td>
<td>CVTRFL</td>
<td>Convert Rounded FFloating to Long</td>
</tr>
<tr>
<td>6A</td>
<td>CVTDL</td>
<td>Convert DFloating to Long</td>
</tr>
<tr>
<td>6B</td>
<td>CVTRDL</td>
<td>Convert Rounded DFloating to Long</td>
</tr>
<tr>
<td>4AFD</td>
<td>CVTGL</td>
<td>Convert GFloating to Long</td>
</tr>
<tr>
<td>4BFD</td>
<td>CVTRGL</td>
<td>Convert Rounded GFloating to Long</td>
</tr>
<tr>
<td>6AFD</td>
<td>CVTHL</td>
<td>Convert HFloating to Long</td>
</tr>
<tr>
<td>6BFD</td>
<td>CVTRHL</td>
<td>Convert Rounded HFloating to Long</td>
</tr>
<tr>
<td>56</td>
<td>CVTFD</td>
<td>Convert F Floating to DFloating</td>
</tr>
<tr>
<td>99FD</td>
<td>CVTFG</td>
<td>Convert F Floating to GFloating</td>
</tr>
<tr>
<td>98FD</td>
<td>CVTFH</td>
<td>Convert F Floating to HFloating</td>
</tr>
<tr>
<td>76</td>
<td>CVTDF</td>
<td>Convert D Floating to FFloating</td>
</tr>
<tr>
<td>32FD</td>
<td>CVTDD</td>
<td>Convert D Floating to HFloating</td>
</tr>
<tr>
<td>33FD</td>
<td>CVTGF</td>
<td>Convert G Floating to FFloating</td>
</tr>
<tr>
<td>56FD</td>
<td>CVTGH</td>
<td>Convert G Floating to HFloating</td>
</tr>
<tr>
<td>F6FD</td>
<td>CVTHF</td>
<td>Convert H Floating to FFloating</td>
</tr>
<tr>
<td>F7FD</td>
<td>CVTHD</td>
<td>Convert H Floating to DFloating</td>
</tr>
<tr>
<td>76FD</td>
<td>CVTHG</td>
<td>Convert H Floating to GFloating</td>
</tr>
</tbody>
</table>
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
- CVTBFG exact
- CVTBDG exact
- CVTBDH exact
- CVTBFHG exact
- CVTBLF exact
- CVTBLD exact
- CVTBLG exact
- CVTBLH exact
- CVTBF truncated
- CVTBD truncated
- CVTBFG truncated
- CVTBDG truncated
- CVTBDH truncated
- CVTBFHG truncated
- CVTBLF truncated
- CVTBLD truncated
- CVTBLG truncated
- CVTBLH truncated
- CVTBF rounded
- CVTBD rounded
- CVTBFG rounded
- CVTBDG rounded
- CVTBDH rounded
- CVTBFHG rounded
- CVTBLF truncated
- CVTBLD truncated
- CVTBLG truncated
- CVTBLH truncated
- CVTWF exact
- CVTWFH exact
- CVTWF rounded
- CVTWFH rounded
- CVTWG exact
- CVTWGH exact
- CVTWH exact
- CVTWHG exact
- CVTWF rounded
- CVTWFH rounded
- CVTWG rounded
- CVTWGH rounded
- CVTWFH rounded

Notes:

1. Only CVTDF, CVTGF, CVTHF, CVTHD, and CVTHG can result in floating overflow fault; the destination operand is unaffected and the condition codes are UNPREDICTABLE.
2. Only converts with a floating point source operand can result in a reserved operand fault. On a reserved operand fault, the destination operand is unaffected and the condition codes are UNPREDICTABLE.

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

4. Only CVTGF, CVTHF, CVTHD, and CVTHG can result in floating underflow. If FU is set a fault occurs. Zero is stored as the result of floating underflow only if FU is clear. On a floating underflow fault, the destination operand is unaffected. If FU is clear, the destination operand is replaced by 0 and no exception occurs.
DIV  Divide

Format:

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

Operation:

quo <- quo / divr;  12 operand
quo <- divd / divr;  13 operand

Condition Codes:

N <- quo LSS 0;
Z <- quo EQL 0;
V <- [floating overflow] or {divr EQL 0};
C <- 0;

Exceptions:

floating overflow
floating underflow
divide by zero
reserved operand

Opcodes:

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

Description:

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

Notes:

1. On a reserved operand fault, the quotient operand is unaffected and the condition codes are UNPREDICTABLE.
2. On floating underflow, if FU is set a fault occurs. Zero is stored as the result of floating underflow only if FU is clear. On a floating underflow fault, the quotient operand is unaffected. If FU is clear, the quotient operand is replaced by 0 and no exception occurs.

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

4. On divide by zero, the quotient operand and condition codes are affected as in 3. above.
EMOD  Extended Multiply and Integerize

Format:

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

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

Operation:

  int <- integer part of muld * {mulr'mulrx};
  fract <- fractional part of muld * {mulr'mulrx};

Condition Codes:

  N <- fract LSS 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 fault occurs. The integer and fraction parts are replaced by zero on the occurrence of floating underflow only if FU is clear. On a floating underflow fault, the integer and fraction parts are unaffected. If FU is clear, the integer and fraction parts are replaced by 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.
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  MNEGFP  Move Negated F Floating
    72  MNEGD  Move Negated D Floating
    52FD MNEGG  Move Negated G Floating
    72FD MNEGH  Move Negated H Floating

Description:

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

Notes:

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

Format:

   opcode src.r, dst.w

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.
MUL Multiply

Format:

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

Operation:

prod <- prod * mulr; 12 operand
prod <- muld * mulr; 13 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
44PD MULG2 Multiply G_floating 2 Operand
45PD MULG3 Multiply G_floating 3 Operand
64PD MULH2 Multiply H_floating 2 Operand
65PD MULH3 Multiply H_floating 3 Operand

Description:

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

Notes:

1. On a reserved operand fault, the product operand is unaffected and the condition codes are UNPREDICTABLE.
2. On floating underflow, if FU is set a fault occurs. Zero is stored as the result of floating underflow only if FU is clear. On a floating underflow fault, the product operand is unaffected. If FU is clear, the product operand is replaced by 0 and no exception occurs.

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

Format:

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

Operation:

tmpl <= degree;
if tmpl GTU 31 then RESERVED OPERAND FAULT;
tmp2 <= tbladdr;
tmp3 <= {(tmp2)+};     !tmp3 accumulates the partial result
                        !tmp3 is of type x
if POLYH then -(SP) <= arg;
while tmpl GTU 0 do
   begin     !computation loop
      tmp4 <= {arg * tmp3};    !tmp4 accumulates new partial result.
                         !tmp3 has old partial result.
                          !Perform multiply, and retain the 31 (POLYF),
                          !63 (POLYD, POLYG), or 127 (POLYH) most significant
                          !bits of the fraction by truncating the unnormalized
                          !product. (The most significant bit of the 31, 63,
                          !or 127 bits in the product magnitude will be zero
                          !if the product magnitude is LSS 1/2 and GEQ 1/4.)
                          !Use the result in the following add operation.
      tmp4 <= tmp4 + (tmp2);
                         !normalize, and round to type x.
                         !Check for over/underflow only after the combined
                         !multiply/add/normalize/round sequence.
      if OVERFLOW then FLOATING OVERFLOW FAULT
      if UNDERFLOW then
         begin
            if FU EQL 1 then FLOATING UNDERFLOW FAULT;
            tmp4 <= 0;    !force result to 0;
         end;
      tmp1 <= tmpl - 1;
tmp2 <= tmp2 + {size of data type};
tmp3 <= tmp4;    !update partial result in tmp3
   end;
if POLYF then
   begin
      R0 <= tmp3;
      R1 <= 0;
      R2 <= 0;
      R3 <= tmp2;
   end;
if POLYD or POLYG then
   begin
      R1'R0 <= tmp3;
      R2 <= 0;
      R3 <= tmp2;
      R4 <= 0;
Instructions 2-Feb-81 -- Rev 6.2
FLOATING POINT INSTRUCTIONS

R5 <- 0;
end;
if POLYH then
begin
SP <- SP + 16;
R3'R2'R1'R0 <- tmp3;
R4 <- 0;
R5 <- tmp2;
end;

Condition Codes:

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

Exceptions:

floating overflow
floating underflow
reserved operand

Opcodes:

55  POLYP  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. If PSL<FPD> = 0, the instruction faults and all relevant side effects are restored to their original state.

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

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

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

   POLYH
   R3'R2'R1'R0 = tmp3 !partial result after iteration prior to the one causing the overflow/underflow
   R4<7:0> = tmp1 !number of iterations remaining
   R4<31:8> = implementation specific
   R5 = tmp2 !points to table entry causing exception
arg is saved on the stack in use during the faulting instruction.

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

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

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

5. On a reserved operand fault:
   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.

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

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

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

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

Example:

To compute \( P(x) = C_0 + C_1 x + C_2 x^2 \)
where \( C_0 = 1.0, \) \( C_1 = .5, \) and \( C_2 = .25 \)
POLYF X,#2,PTABLE

PTABLE: .FLOAT 0.25 ;C2
       .FLOAT 0.5  ;C1
       .FLOAT 1.0  ;C0
SUB      Subtract

Format:

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

Operation:

dif <- dif - sub;            12 operand
dif <- min - sub;            13 operand

Condition Codes:

N <- dif LSS 0;
Z <- dif EQ 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 fault occurs. Zero is stored as the result of floating underflow only if FU is clear. On a floating underflow fault, the difference operand is unaffected. If FU is clear, the difference operand is replaced by 0 and no exception occurs.

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

Format:

opcode src.rx

Operation:

src - 0;

Condition Codes:

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

Exceptions:

reserved operand

Opcodes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>TSTF</td>
<td>Test F_floating</td>
</tr>
<tr>
<td>73</td>
<td>TSTD</td>
<td>Test D_floating</td>
</tr>
<tr>
<td>53FD</td>
<td>TSTG</td>
<td>Test G_floating</td>
</tr>
<tr>
<td>73FD</td>
<td>TSTH</td>
<td>Test H_floating</td>
</tr>
</tbody>
</table>

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.
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 interruption, the instruction resumes transparently. The format of the control block is:

```
+-------------------------+  
|  | LENGTH 1 |  
|  | : R0     |  
| ADDRESS 1              |  
|  | LENGTH 2 |  
|  | : R1     |  
| ADDRESS 2              |  
|  | LENGTH 3 |  
|  | : R2     |  
| ADDRESS 3              |  
```

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

Memory access faults will not occur when a zero length string is specified because no memory reference occurs.
The following instructions are described in this section.

1. Compare Characters 3 Operand
   CMPC3 len.rw, src1addr.ab, src2addr.ab, {R0-3.w1}

2. Compare Characters 5 Operand
   CMPC5 src1len.rw, src1addr.ab, fill.rb, src2len.rw, src2addr.ab, {R0-3.w1}

3. Locate Character
   LOCC char.rb, len.rw, addr.ab, {R0-1.w1}

4. Match Characters
   MATCHC len1.rw, addr1.ab, len2.rw, addr2.ab, {R0-3.w1}

5. Move Character 3 Operand
   MOVC3 len.rw, srcaddr.ab, dstaddr.ab, {R0-5.w1}

6. Move Character 5 Operand
   MOVC5 srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab, {R0-5.w1}

7. Move Translated Characters
   MOVTC srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab, {R0-5.w1}

8. Move Translated Until Character
   MOVTUC srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen, rw, dstaddr.ab, {R0-5.w1}

9. Scan Characters
   SCANC len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.w1}

10. Skip Character
    SKPC char.rb, len.rw, addr.ab, {R0-1.w1}

11. Span Characters
    SPANC len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.w1}
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:

tmpl1 <- len;  !3 operand
tmpl2 <- src1addr;

tmpl3 <- src2addr;
if tmpl1 EQL 0 then; !Condition Codes affected on tmpl1 EQL 0
if tmpl1 GTRU 0 then
   begin

   while {tmpl1 NEQU 0} do
   if (tmpl2) EQL (tmpl3) then
      !Condition Codes affected on ((tmpl2) EQL (tmpl3))
         begin
            tmpl1 <- tmpl1 - 1;
            tmpl2 <- tmpl2 + 1;
            tmpl3 <- tmpl3 + 1;
            end;
   else exit while loop;
         end;
   end;
   R0 <- tmpl1;
   R1 <- tmpl2;
   R2 <- R0;
   R3 <- tmpl3;

tmpl1 <- src1len;
tmpl2 <- src1addr;
tmpl3 <- src2len;
tmpl4 <- src2addr;

if {tmpl1 EQL 0} AND {tmpl3 EQL 0} then;
   !Condition codes affected on {tmpl1 EQL 0} AND {tmpl3 EQL 0}
while {tmpl1 NEQU 0} AND {tmpl3 NEQU 0} do
   if (tmpl2) EQL (tmpl4) then
      !Condition Codes affected on ((tmpl2) EQL (tmpl4))
         begin
            tmpl1 <- tmpl1 - 1;
            tmpl2 <- tmpl2 + 1;
            tmpl3 <- tmpl3 - 1;
         end;
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.
LOCC  Locate Character

Format:

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

Operation:

    tmpl ← len;
    tmp2 ← addr;
    if tmpl GTRU 0 then
        begin
        while {tmpl NEQ 0} AND {(tmp2) NEQ char} do
            begin
                tmpl ← tmpl - 1;
                tmp2 ← tmp2 + 1;
            end;
        end;
    R0 ← tmpl;
    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.
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;

if {tmp3 LSSU tmp1} then
begin
  tmp4 <- tmp4 + tmp3;
tmp3 <- 0;
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 i.e. objaddr + objlen; otherwise the address of the object string.
   
   R2 = if a match occurred, the number of bytes remaining in the source string; otherwise 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 i.e. srcaddr + srclen.

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

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

3. If the source string has zero length and the object string has non-zero length, condition code Z is cleared and registers R0-R3 are left just as though the substring were not found.
MOV C    Move Character

Format:

opcode len.rw, srcaddr.ab, dstaddr.ab    3 operand
opcode src.len.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab    5 operand

Operation:

tmp1 <- len;
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;
    end;
R0 <- 0;
R2 <- 0;
R4 <- 0;
R5 <- 0;
Instructions 12-Feb-82 -- Rev 7

CHARACTER STRING INSTRUCTIONS

tmp1 <- src.len;
tmp2 <- src.addr;
tmp3 <- dst.len;
tmp4 <- dst.addr;
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;
  end;
R0 <- tmp1;
R2 <- 0;
R4 <- 0;
R5 <- 0;
Character String Instructions

Condition Codes:

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

N <- srcLen LSS dstLen; !MVC5  
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 MOV C3:
   
   \[ R0 = 0 \]
   
   \[ R1 = \text{address of one byte beyond the source string} \]
   
   \[ R2 = 0 \]
   
   \[ R3 = \text{address of one byte beyond the destination string.} \]
   
   \[ R4 = 0 \]
   
   \[ R5 = 0 \]

2. After execution of MOV C5:
   
   \[ R0 = \text{number of unmoved bytes remaining in source string.} \]
   \[ R0 \text{ is non-zero only if source string is longer than destination string} \]
   
   \[ R1 = \text{address of one byte beyond the last byte in source string that was moved} \]
   
   \[ R2 = 0 \]
   
   \[ R3 = \text{address of one byte beyond the destination string.} \]
   
   \[ R4 = 0 \]
   
   \[ R5 = 0 \]

3. MOV C3 is the preferred way to copy one block of memory to another.

4. MOV C5 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
  {tmp1 NEQU 0} AND {tmp3 NEQU 0}
  while {tmp4 NEQU 0}
      (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 table, the destination string and registers R0 through R5 are UNPREDICTABLE. If the source and destination strings overlap and their addresses are not identical, the destination string and registers R0 through R5 are UNPREDICTABLE. If the source and destination string addresses are identical, the translation is performed correctly.

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.
SCANC  Scan Characters

Format:

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

Operation:

\[
\begin{align*}
tmp1 & \leftarrow len; \\
tmp2 & \leftarrow addr; \\
\text{if } tmp1 & \text{ GTRU } 0 \text{ then} \\
& \begin{align*}
& \text{begin} \\
& \{ \{ \text{tbladdr + ZEXT((tmp2)) AND mask} \} \text{ EQ } 0 \} \text{ do} \\
& \begin{align*}
& \text{begin} \\
& \text{tmp1 } \leftarrow \text{tmp1 } - 1; \\
& \text{tmp2 } \leftarrow \text{tmp2 } + 1; \\
& \text{end} \\
& \text{end} \\
& \text{end} \\
& \text{R0 } \leftarrow \text{tmp1}; \\
& \text{R1 } \leftarrow \text{tmp2}; \\
& \text{R2 } \leftarrow \emptyset; \\
& \text{R3 } \leftarrow \text{tbladdr}; \\
\end{align*}
\end{align*}
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow \emptyset; \\
Z & \leftarrow \text{R0 EQ } 0; \\
V & \leftarrow \emptyset; \\
C & \leftarrow \emptyset; \\
\end{align*}
\]

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.
SKPC     Skip Character

Format:

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

Operation:

    tmpl <- len;
    tmp2 <- addr;
    if tmpl GTRU 0 then
      begin
        while {tmpl NEQ 0} AND {(tmp2) EQL char} do
          begin
            tmpl <- tmpl - 1;
            tmp2 <- tmp2 + 1;
          end;
      end;
    end;
    R0 <- tmpl;
    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 unequal
        one) if unequal 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 equal to character.
SPANC  Span Characters

Format:

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

Operation:

tmpl1 <- len;
tmp2 <- addr;
if tmpl1 GTRU 0 then
    begin
        while {tmpl1 NEQU 0} AND
            {[(tbladdr + ZEXT((tmp2))) AND mask] NEQ 0} do
            begin
                tmpl1 <- tmpl1 - 1;
tmp2 <- tmpl2 + 1;
            end;
        end;
R0 <- tmpl1;
R1 <- tmpl2;
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.
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.
CYCLIC REDUNDANCY CHECK INSTRUCTION

CRC Calculate Cyclic Redundancy Check

Format:

opcode tbl.ab, inicrc.r1, 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)));
  tmlpl <- 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:

\[ \text{polyn}^{<n>} \leftarrow \{\text{coefficient of } x^{<\text{order}-1-n}>\} \]

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.

```plaintext
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
CONTINUE
150 TABLE(INDEX) = TMP

CONTINUE
190 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^9+x^7+x^5+x^4+x^2+x+1
poly: EDB8320 (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:

<table>
<thead>
<tr>
<th>shift</th>
<th>steps per byte</th>
<th>table index (s)</th>
<th>table index (l)</th>
<th>use table entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>tmp3&lt;0&gt;</td>
<td>8</td>
<td>[0]=0,[8]</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>tmp3&lt;1:0&gt;</td>
<td></td>
<td>[0]=0,[4],[8],[12]</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>tmp3&lt;3:0&gt;</td>
<td>1</td>
<td>all</td>
</tr>
</tbody>
</table>

6. If the stream has zero length, R0 receives the initial CRC.
4.12 DECIMAL STRING INSTRUCTIONS

Decimal string instructions operate on Packed Decimal strings. Convert instructions are provided between Packed Decimal and Trailing Numeric String (Overpunched and Zoned) and Leading Separate Numeric string formats. Where necessary a specific data type is identified. Where the phrase decimal string is used, it means any of the three data types.

A decimal string is specified by 2 operands:

1. For all decimal strings the length is the number of digits in the string. The number of bytes in the string is a function of the length and the type of decimal string referenced (see Chapter 2).

2. The address of the lowest addressed byte of the string. This byte contains the most significant digit for Trailing Numeric, and packed decimal strings. This byte contains a sign for Left Separate Numeric strings. The address is specified by a byte operand of address access type.

Each of the decimal string instructions uses general registers 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:

```
  3 1 0
   |   |   +
   |   +   +
   |       +
   +------+
   |       +
   |       +
   +-------+
       |     +
   |       +
   +-------+
       |     +
   |       +
   +-------+
       |     +
   |       +
   +-------+
       |     +
   |       +
   +-------+
       |     +
   |       +
   +-------+
       |     +
   |       +
   +-------+
   |       +
   |       +
   +-------+
   |       +
   |       +
   +-------+
```

The fields ADDRESS 1, ADDRESS 2 and ADDRESS 3 (if required) contain the address of the byte containing the most significant digit of the first, second and third (if required) string operands respectively.
The decimal string instructions treat decimal strings as integers with the decimal point assumed immediately beyond the least significant digit of the string. If a string in which a result is to be stored is longer than the result, its most significant digits are filled with zeros.

4.12.1 Decimal Overflow

Decimal overflow occurs if the destination string is too short to contain all the digits (excluding leading zeroes) of the result. On overflow, the destination string is replaced by the correctly signed least significant digits of the true result (even if the stored result is -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, except for CVTPT which does not fixup a -0 to a +0. However, when digits are lost because of overflow, a zero result receives the sign (positive or negative) of the correct result.

A decimal string with value -0 is treated as identical to a decimal string with value +0. Thus for example +0 compares 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 abort 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. The PC points to the opcode of the instruction causing the exception.

4.12.4 UNPREDICTABLE Results

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

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

Packed decimal strings generated by the decimal string instructions always have the preferred sign representation: 12 for "+" and 13 for "-". An even length packed decimal string is always generated with a "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 value of a zero length trailing numeric string is identically 0.

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 abort will occur if an invalid sign is detected. The value of a zero length leading separate numeric string is identically 0.
4.12.7 Instruction Descriptions

The following instructions are described in this section.

1. Add Packed 4 Operand
   ADDP4 addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab, [R0-3.wl]

2. Add Packed 6 Operand
   ADDP6 addlen.rw, addladdr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab, [R0-5.wl]

3. Arithmetic Shift and Round Packed
   ASHP cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab, [R0-3.wl]

4. Compare Packed 3 Operand
   CMFPJ len.rw, src1addr.ab, src2addr.ab, [R0-3.wl]

5. Compare Packed 4 Operand
   CMFP4 src1len.rw, src1addr.ab, src2len.rw, src2addr.ab, [R0-3.wl]

6. Convert Long to Packed
   CVTLP src.rl, dstlen.rw, dstaddr.ab, [R0-3.wl]

7. Convert Packed to Long
   CVTPL srclen.rw, srcaddr.ab, [R0-3.wl], dst.wl

8. Convert Packed to Leading Separate
   CVTLP srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab, [R0-3.wl]

9. Convert Packed to Trailing
   CVTPT srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab, [R0-3.wl]

10. Convert Leading Separate to Packed
    CVTSP srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab, [R0-3.wl]

11. Convert Trailing to Packed
    CVTTP srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab, [R0-3.wl]

12. Divide Packed
    DIVP divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quolen.rw, quoadr.ab, [R0-5.wl, -16(SP):-1(SP).wb]

13. Move Packed
    MOV P len.rw, srcaddr.ab, dstaddr.ab, [R0-3.wl]

14. Multiply Packed
    MULP mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab, [R0-5.wl]
15. Subtract Packed 4 Operand
   SUBP4 sublen.rw, subaddr.ab, diflen.rw, difaddr.ab, {R0-3.w1}

16. Subtract Packed 6 Operand
   SUBP6 sublen.rw, subaddr.ab, minlen.rw, minaddr.ab,
   diflen.rw, difaddr.ab, {R0-5.w1}
ADD P Add Packed

Format:

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

Operation:

{sumaddr + ZEXT(sumlen/2)} : sumaddr <=
{addaddr + ZEXT(addlen/2)} : addaddr; !4 operand

{sumaddr + ZEXT(sumlen/2)} : sumaddr <=
{add2addr + ZEXT(add2len/2)} : add2addr +
{addladdr + ZEXT(add1len/2)} : addladdr; !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:
   \[ R0 = 0 \]
   \[ R1 = \text{address of the byte containing the most significant digit of the addend string} \]
   \[ R2 = 0 \]
   \[ R3 = \text{address of the byte containing the most significant digit of the sum string} \]

2. After execution of ADDP6:
   \[ R0 = 0 \]
   \[ R1 = \text{address of the byte containing the most significant digit of the addend1 string} \]
   \[ R2 = 0 \]
   \[ R3 = \text{address of the byte containing the most significant digit of the addend2 string} \]
   \[ R4 = 0 \]
   \[ R5 = \text{address of the byte containing the most significant digit of the sum string} \]

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 abort occurs.
ASHP    Arithmetic Shift and Round Packed

Format:

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

Operation:

((dstaddr + ZEXT(dstlen/2)) : dstaddr) <-
  (((srcaddr + ZEXT(srcalen/2)) : srcaddr)  
   + [round <3:0>*[10 ** [-cnt-1]]])        
   * [10 ** cnt];

Condition Codes:

N <- {dst string} LSS Ø;
Z <- {dst string} EQL Ø;
V <- {decimal overflow};
C <- Ø;

Exceptions:

reserved operand
decimal overflow

Opcodes:

F8    ASHP    Arithmetic Shift and Round Packed

Description:

The source string specified by the source length and source address operands is scaled by a power of 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 = Ø

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

   R2 = Ø
R3 = address of the byte containing the most significant digit of the destination string

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

3. 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.
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:

\[
\begin{align*}
(src1addr + \text{ZEXT}(\text{len}/2)) : src1addr - \\
(src2addr + \text{ZEXT}(\text{len}/2)) : src2addr;  \quad \text{3 operand}
\end{align*}
\]

\[
\begin{align*}
(src1addr + \text{ZEXT}(\text{src1len}/2)) : src1addr - \\
(src2addr + \text{ZEXT}(\text{src2len}/2)) : src2addr;  \quad \text{4 operand}
\end{align*}
\]

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 Operands
- 37  CMPP4  Compare Packed 4 Operands

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 abort occurs.
CVTLP    Convert Long to Packed

Format:

opcode src.r1, 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 abort.

3. Overlapping operands produce correct results.
CVTPL  Convert Packed to Long

Format:

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

Operation:

    dst <- conversion of ({srcaddr + ZEXT(srclen/2)} : srcaddr);

Condition Codes:

    N <- dst LSS 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 abort or if the string contains an invalid nibble.

3. The destination operand is stored after the registers are updated as specified in 1 above. Thus 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.
CVTPS  Convert Packed to Leading Separate Numeric

Format:

opcode src.len.rw, src.addr.ab, dst.len.rw, dst.addr.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 abort occurs.

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

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

5. If the conversion produces a -0 without overflow, the destination leading separate numeric string is changed to a +0 representation.
CVTP T  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  CVTP T  Convert Packed to Trailing Numeric

Description:

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

Conversion is effected by using the highest addressed byte (even if the source string value is -0) 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 abort occurs.

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

4. By appropriate specification of the table, conversion to any form of trailing numeric string may be realized. See Chapter 2 for the preferred form of trailing overpunch, zoned and unsigned data. In addition, the table may be set up for absolute value, negative absolute value or negated conversions. The translation table may be referenced even if the length of the destination string is zero.

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

6. If decimal overflow occurs, the value stored in the destination may be different from the value indicated by the condition codes (Z and N bits).
CVTSP  Convert Leading Separate Numeric to Packed

Format:

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

Operation:

{dst string} <- conversion of {src string}

Condition Codes:

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

Exceptions:

reserved operand
decimal overflow

Opcodes:

09  CVTSP  Convert Leading Separate Numeric to Packed

Description:

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

Notes:

1. A reserved operand abort occurs if:

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 abort occurs.
CVTTP   Convert Trailing Numeric to Packed

Format:

opcode src.len.rw, src.addr.ab, tbl.addr.ab, dst.len.rw, dst.addr.ab

Operation:

{dst string} <- conversion of {src string}

Condition Codes:

N <- {dst string}LSB 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 abort 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 abort occurs.

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

5. If the length of the source string is 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.
DIVP  Divide Packed

Format:

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

Operation:

({quoaddr + ZEXT(quolen/2)} : quoaddr) <-
    ({divdaddr + ZEXT(divdlen/2)} : divdaddr) /
    ({divraddr + ZEXT(divrlen/2)} : divraddr);

Condition Codes:

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

Exceptions:

reserved operand
decimal overflow
divide by zero

Opcodes:

27  DIVP  Divide Packed

Description:

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

Notes:

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

2. The division is performed such that:

   1. The absolute value of the remainder (which is lost) is less
      that the absolute value of the divisor.

   2. The product of the absolute value of the quotient times the
      absolute value of the divisor is less than or equal to the
      absolute value of the dividend.
3. The sign of the quotient is determined by the rules of algebra from the signs of the dividend and the divisor. If the value of the quotient is zero, the sign is always positive.

3. After execution:
   
   R0 = 0

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

   R2 = 0

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

   R4 = 0

   R5 = address of the byte containing the most significant digit of the quotient string.

4. The quotient string, R0 through R5, and the condition codes are UNPREDICTABLE if the quotient string overlaps the divisor or dividend strings, the divisor or dividend string contains an invalid nibble, the divisor is 0 or a reserved operand abort occurs.
MOVP    Move Packed

Format:

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

Operation:

\[
(dstaddr + \text{ZEXT}(|len|/2)) : \text{dstaddr} \leftarrow
(srcaddr + \text{ZEXT}(|len|/2)) : \text{srcaddr};
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow \text{dst string \ LSS \ 0;} \\
Z & \leftarrow \text{dst string \ EQL \ 0;} \\
V & \leftarrow \emptyset; \\
C & \leftarrow C;
\end{align*}
\]

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:

\[
R0 = \emptyset
\]

\[
R1 = \text{address of the byte containing the most significant digit of the source string}
\]

\[
R2 = \emptyset
\]

\[
R3 = \text{address of the byte containing the most significant digit of the destination string.}
\]

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 abort occurs.
3. If the source is -0, the result is +0, N is cleared and Z is set.
MULP    Multiply Packed

Format:

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

Operation:

  (((prodaddr + ZEXT(prodlen/2)) : prodaddr) <=
      ((muldaddr + ZEXT(muldlen/2)) : muldaddr) *
      ((mulraddr + ZEXT(mulrlen/2)) : mulraddr);

Condition Codes:

N <- {prod string} LSS 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 abort occurs.
SUBP  Subtract Packed

Format:

```
opcode sublen.rw, subaddr.ab, diflen.rw, difaddr.ab       4 operand

opcode sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, diflen.rw, difaddr.ab 6 operand
```

Operation:

```
{difaddr + ZEXT(diflen/2)} : difaddr) <-
{{difaddr + ZEXT(diflen/2)} : difaddr) -
{{subaddr + ZEXT(sublen/2)} : subaddr}; !4 operand

{difaddr + ZEXT(diflen/2)} : difaddr) <-
{{minaddr + ZEXT(minlen/2)} : minaddr) -
{{subaddr + ZEXT(sublen/2)} : subaddr}; !6 operand
```

Condition Codes:

```
N <- {dif string} LSS 0;
Z <- {dif string} EQL 0;
V <- {decimal overflow};
C <- 0;
```

Exceptions:

reserved operand
decimal overflow

Opcodes:

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

Description:

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

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

1. After execution of SUBP4:
   
   \[ R0 = 0 \]
   
   \[ R1 = \text{address of the byte containing the most significant digit of the subtrahend string} \]
   
   \[ R2 = 0 \]
   
   \[ R3 = \text{address of the byte containing the most significant digit of the difference string} \]

2. After execution of SUBP6:
   
   \[ R0 = 0 \]
   
   \[ R1 = \text{address of the byte containing the most significant digit of the subtrahend string} \]
   
   \[ R2 = 0 \]
   
   \[ R3 = \text{address of the byte containing the most significant digit of the minuend string} \]
   
   \[ R4 = 0 \]
   
   \[ R5 = \text{address of the byte containing the most significant digit of the difference string} \]

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 abort occurs.
4.13 EDIT INSTRUCTION

This instruction is designed to implement the common editing functions which occur in handling fixed format output. It operates by converting a packed decimal string to a character string. This operation is exemplified by a MOVE to a numeric edited (PICTURE) item in COBOL or PL/I, but the instruction can be used for other applications as well. The operation consists of converting an input packed decimal number to an output character string, generating characters for the output. When converting digits, options include leading zero fill, leading zero protection, insertion of floating sign, insertion of floating currency symbol, insertion of special sign representations, and blanking an entire field when it is zero.

The operands to the EDITPC instruction are an input packed decimal string descriptor, a pattern specification, and the starting address of the output string. The packed decimal descriptor is a standard VAX-11 operand pair of the length of the decimal string in digits (up to 31) and the starting address of the string. The pattern specification is the starting address of a pattern operation editing sequence which is interpreted much the way that the normal instructions are. The output string is described by only its starting address because the pattern defines the length unambiguously.

While the EDITPC instruction is operating, it manipulates two character registers and the four condition codes. One character register contains the fill character. This is normally an ASCII blank, but would be changed to asterisk for check protection. The other character register contains the sign character. Initially this contains either an ASCII blank or a minus sign depending upon the sign of the input. This can be changed to allow other sign representations such as plus/minus or plus/blank and can be manipulated in order to output special notations such as CR or DB. The sign register can also be changed to the currency sign in order to implement a floating currency sign. After execution, the condition codes contain the sign of the input (N), the presence of a zero source (Z), an overflow condition (V), and the presence of significant digits (C). Condition code N is determined at the start of the instruction and is not changed thereafter (except for correcting a -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 abort};
PSW<Y>C< > <- 0;
PSW<Z> <- 1;
PSW<N> <- {src has minus sign};
R0 <- srclen;
tmp1 <- R0;
R1 <- srcaddr;
R2 <- ??? ' {if PSW<N> EQ 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> EQ 1 then PSW<N> <- 0;

Condition Codes:

N <- {src string} LSS 0;  !N <- 0 if src is -0
Z <- {src string} EQ 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 edited version of the source string specified by the source length and source address operands. The editing is performed according to the pattern string starting at the address pattern and extending until a pattern end (EOS$END) pattern operator is encountered. The pattern string consists of one byte pattern operators. Some pattern operators take no operands. Some take a repeat count which is contained in the rightmost nibble of the pattern operator itself. The rest take a one byte operand which follows the pattern operator immediately. This operand is either an unsigned integer length or a byte character. The individual pattern operators are described on the following pages.

Notes:

1. A reserved operand abort occurs if src.len GTRU 31.

2. The destination string is UNPREDICTABLE if the source string contains an invalid nibble, if the EOS$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 EOS$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:

\[
\begin{align*}
N &= \text{src has minus sign} \\
Z &= \text{all source digits 0 so far} \\
V &= \text{non-zero digits lost} \\
C &= \text{significance} \\
R0 &= \text{zeros}\langle15:0\rangle \ ' \ remaining \ src\langle15:0\rangle \\
R1 &= \text{current source location} \\
R2 &= ?? \ ' \ sign \ ' \ fill \\
R3 &= \text{location of edit pattern operator causing exception}
\end{align*}
\]
R4 = ???
R5 = location of next destination byte

where:

zeros = count of source zeros to supply
sign = current contents of sign character register
fill = current contents of fill character register
## Summary of EDIT pattern operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Operand</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0$INSERT</td>
<td>ch</td>
<td>insert character, fill if insignificant</td>
</tr>
<tr>
<td>E0$STORE_SIGN</td>
<td>-</td>
<td>insert sign</td>
</tr>
<tr>
<td>E0$FILL</td>
<td>r</td>
<td>insert fill</td>
</tr>
<tr>
<td><strong>Move</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0$MOVE</td>
<td>r</td>
<td>move digits, filling insignificant</td>
</tr>
<tr>
<td>E0$FLOAT</td>
<td>r</td>
<td>move digits, floating sign</td>
</tr>
<tr>
<td>E0$END_FLOAT</td>
<td></td>
<td>end floating sign</td>
</tr>
<tr>
<td><strong>Fixup</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0$BLANK_ZERO</td>
<td>len</td>
<td>fill backward when zero</td>
</tr>
<tr>
<td>E0$REPLACE_SIGN</td>
<td>len</td>
<td>replace with fill if -0</td>
</tr>
<tr>
<td><strong>Load</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0$LOAD_FILL</td>
<td>ch</td>
<td>load fill character</td>
</tr>
<tr>
<td>E0$LOAD_SIGN</td>
<td>ch</td>
<td>load sign character</td>
</tr>
<tr>
<td>E0$LOAD_PLUS</td>
<td>ch</td>
<td>load sign character if positive</td>
</tr>
<tr>
<td>E0$LOAD_MINUS</td>
<td>ch</td>
<td>load sign character if negative</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0$SET_SIGNIF</td>
<td>-</td>
<td>set significance flag</td>
</tr>
<tr>
<td>E0$CLEAR_SIGNIF</td>
<td>-</td>
<td>clear significance flag</td>
</tr>
<tr>
<td>E0$ADJUST_INPUT</td>
<td>len</td>
<td>adjust source length</td>
</tr>
<tr>
<td>E0$END</td>
<td>-</td>
<td>end edit</td>
</tr>
</tbody>
</table>

**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  \n41  EO$LOAD_SIGN  |
42  EO$LOAD_PLUS  |-- character is in next byte
43  EO$LOAD_MINUS  |
44  EO$INSERT  /
45  EO$BLANK_ZERO  \n46  EO$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  \n91..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 EQL 0 then {reserved operand};
   if R0 LSS 0 then
      begin
      READ <- 0;
      R0<31:16> <- R0<31:16> + 1;   !see EOSADJUST_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>
Instructions

EDIT INSTRUCTION

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$SET_SIGNIP or EO$END_FLOAT).
EDITOR INSTRUCTION

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.
Instructions

EDIT INSTRUCTION

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 NEQ 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.
Instructions

EDIT INSTRUCTION

EO$FLOAT		Float Sign

Purpose:
Move digits, floating the sign across insignificant digits

Format:

\[ \text{pattern } r \]

Operation:

\[
\text{repeat } r \text{ do } \\
\quad \text{begin} \\
\quad \text{tmp } <- \text{ READ;} \\
\quad \text{if tmp } \neq \text{ 0 then} \\
\quad \quad \text{begin} \\
\quad \quad \quad \text{if PSW}\langle C \rangle \text{ EQL 0 then} \\
\quad \quad \quad \quad \text{begin} \\
\quad \quad \quad \quad \text{STORE}(\text{sign}); \\
\quad \quad \quad \quad \text{PSW}\langle Z \rangle <- 0; \\
\quad \quad \quad \quad \text{PSW}\langle C \rangle <- 1; \! \text{ set significance} \\
\quad \quad \quad \text{end;} \\
\quad \quad \quad \text{if PSW}\langle C \rangle \text{ EQL 0 then STORE(}\text{fill}) \\
\quad \quad \quad \text{else STORE("0" + tmp);} \\
\quad \quad \text{end;} \\
\quad \text{end;} \\
\]

Pattern operators:

\[ \text{Ax } \text{ EO$FLOAT } \text{ Float Sign} \]

Description:
The right nibble of the pattern operator is the repeat count. For repeat times, the following algorithm is executed. The next digit from the source is examined. If it is non-zero and significance is not yet set, then the contents of the sign register is stored in the destination, significance is set, and zero is cleared. If the digit is significant, it is stored in the destination, otherwise the contents of the fill register is stored in the destination.

Notes:

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

2. This pattern operator is used to move digits with a floating arithmetic sign. The sign must already be setup as for EO$STORE $SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.
3. This pattern operator is used to move digits with a floating currency sign. The sign must already be setup with a EO$LOAD_SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLS. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.
EO$END_FLOAT End Floating Sign

Purpose:
End a floating sign operation

Format:

pattern

Operation:

if PSW$C = EQL 0 then
  begin
    STORE(sign);
    PSW$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.
EOSBLANK_ZERO    Blank Backwards When Zero

Purpose:
Fixup the destination to be blank when the value is zero

Format:
    pattern   len

Operation:
    if len EQU 0 then {UNPREDICTABLE};
    if PSW<Z> EQL 1 then
       begin
           R5 <- R5 - len;
           repeat len do STORE(fill);
       end;

Pattern operators:

   45    EOSBLANK_ZERO    Blank Backwards When Zero

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

Notes:
1. The length must be non-zero and within the destination string
   already produced. If it is not, the contents of the destination string
   and the memory preceding it are UNPREDICTABLE.
2. This pattern operator is used to blank out any characters
   stored in the destination under a forced significance, such as
   a sign or the digits following the radix point.
EO$REPLACE_SIGN Replace Sign When Zero

Purpose:
Fix up the destination sign when the value is zero

Format:
  pattern   len

Operation:
  if len EQLU Ø then {UNPREDICTABLE};
  if PSW<Z> EQL 1 then (R5 - len) ← fill;

Pattern operators:
  46   EO$REPLACE_SIGN Replace Sign When Zero

Description:
The pattern operator is followed by an unsigned byte integer length. If
the value of the source string is zero (i.e., if Z is set), then the
contents of the fill register is stored into the byte of the destination
string which is length bytes before the current position.

Notes:

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

2. This pattern operator can be used to correct a stored sign
   (EO$SEND_FLOAT or EO$STORE_SIGN) if a minus was stored and the
   source value turned out to be zero.
Instructions
EDIT INSTRUCTION

EO$LOAD_  Load Register

Purpose:
Change the contents of the fill or sign register

Format:

\[
\text{pattern} \quad \text{ch}
\]

Operation:

\[
\begin{align*}
\text{fill} & \leftarrow \text{ch}; \\
\text{sign} & \leftarrow \text{ch};
\end{align*}
\]

!EO$LOAD_FILL

if P$WN \equiv 0 \text{ then sign} \leftarrow \text{ch};

!EO$LOAD_SIGN

if P$WN \equiv 1 \text{ then sign} \leftarrow \text{ch};

!EO$LOAD_PLUS

!EO$LOAD_MINUS

Pattern operators:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>EO$LOAD_FILL Load Fill Register</td>
</tr>
<tr>
<td>41</td>
<td>EO$LOAD_SIGN Load Sign Register</td>
</tr>
<tr>
<td>42</td>
<td>EO$LOAD_PLUS Load Sign Register If Plus</td>
</tr>
<tr>
<td>43</td>
<td>EO$LOAD_MINUS Load Sign Register If Minus</td>
</tr>
</tbody>
</table>

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<#> <- 0;  !EO$CLEAR_SIGNIF
PSW<#> <- 1;  !EO$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;
                        PSW<V> <- 1;
                        end;
                    else R0<31:16> <- R0<15:0> - len;
                        !negative of number to fill
                end;
    end;

Pattern operators:

47    EO$ADJUST_INPUT Adjust Input Length

Description:
The pattern operator is followed by an unsigned byte integer length in the range 1 through 31. If the source string has more digits than this length, the excess leading digits are read and discarded. If any discarded digits are non-zero then overflow is set, significance is set, and zero is cleared. If the source string has fewer digits than this length, a counter is set of the number of leading zeros to supply. This counter is stored as a negative number in 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:

ØØ 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 -Ø, the N condition code is cleared.
4.14 OTHER VAX-11 INSTRUCTIONS

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

1. Chapter 5:
   Probe {Read, Write} Accessability
   PROBE[R,W] mode.rb, len.rw, base.ab

2. Chapter 6:
   Change Mode
   CHM[K,E,S,U] param.rw, {-(ySP).w*}
   Where y=MINU(x, PSL<current_mode>)
   Return from Exception or Interrupt
   REI {(SP)+.r*}

3. Chapter 7:
   Load Process Context
   LDPCTX [PCB.r*, -(KSP).w*]
   Save Process Context
   SVPCTX {(SP)+.r*, PCB.w*}

4. Chapter 9:
   Move To Process Register
   MTPR src.rl, procreg.rl
   Move From Processor Register
   MFPR procreg.rl, dst.wl
Bugcheck

Format:

opcode message.bx

Operation:

{fault to report error}

Condition Codes:

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

Exceptions:

reserved instruction

Opcodes:

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

Description:

The hardware treats these opcodes as RESERVED to DIGITAL and faults. The VAX/VMS operating system treats these as requests to report software detected errors. The in-line message identifier is zero extended to a longword (BUGW) and interpreted as a condition value (see Appendix C, VAX/VMS Run Time Library Reference Manual). If the process is privileged to report bugs, a log entry is made. If the process is not privileged, a reserved instruction is signalled.
CHAPTER 5
MEMORY MANAGEMENT

17-Jun-81 -- Rev 5.3

5.1 INTRODUCTION

Memory management consists of the hardware and software which control the allocation and use of physical memory. Typically, in a multiprogramming system, several processes may reside in physical memory at the same time. The VAX-11 uses memory protection and multiple address spaces to ensure that one process will not affect other processes or the operating system.

To further improve software reliability, four hierarchical access modes provide memory access control. They are, from most to least privileged: kernel, executive, supervisor, and user. Protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write for each of the four access modes. Any location accessible to one mode is also accessible to all more privileged modes. Furthermore, for each access mode, any location that can be written can also be read.

The CPU generates virtual addresses when an image is executed. However, before these addresses can be used to access instructions and data, they must be translated into physical addresses. Memory management software maintains tables of mapping information (page tables) that keep track of where each 512-byte virtual page is located in physical memory. The CPU utilizes this mapping information when it translates virtual addresses to physical addresses.

Therefore, memory management is the scheme that provides both the memory protection and memory mapping mechanisms of the VAX-11. The memory management meets several development goals:

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

2. Allow data structures up to one gigabyte.
3. Provide convenient and efficient sharing of instructions and data.

4. Contribute to software reliability.

A virtual memory system provides a large address space, yet allows programs to run on hardware with small memory configurations. Programs execute in an environment termed a process. The virtual memory system for VAX-11 provides each process with a 4 billion byte address space.

The virtual address space is divided into two equal size spaces, the system address space and the per-process address space. The system address space is the same for all processes. It contains the operating system which is written as callable procedures. Thus all system code can be available to all other system and user code via a simple CALL. Each process has its own separate process address space. However, several processes may have access to the same page, thus providing controlled sharing.

5.2 VIRTUAL ADDRESS SPACE

A virtual address is a 32 bit unsigned integer specifying a byte location in the address space. The programmer sees a linear array of 4,294,967,296 bytes. The virtual address space is broken into 512 byte units termed pages. The page is the unit of relocation and protection.

This virtual address space is too large to be contained in any presently available main memory. Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. Memory management also provides page protection between processes. The operating system controls the virtual-to-physical address mapping tables, and swaps the inactive but used parts of the virtual address space onto the external storage media.

The virtual address space is divided into two parts. The half with the smaller addresses, known as "per-process space," is distinct for each process running on the system. The half with the larger addresses, known as "system space," is shared by all processes. Virtual address space is illustrated in Figure 5-1.
Figure 5-1
Virtual Address Space
5.2.1 Process Space

The smaller-addressed half (addresses 00000000-7FFFFFFF, hex) of the virtual address space is termed "per-process space." The per-process space is divided into two equal parts, the program region (P0 region) and the control region (P1 region). Each process has a separate address translation map for per-process space, so the per-process spaces of all processes are completely disjoint (see the section on Sharing at the end of this chapter). The address map for per-process space is context switched (changed) when the process running on the system is changed (see the chapter on Process Structure).

5.2.2 System Space

The larger-addressed half (addresses 80000000-FFFFFFFE, hex) of the virtual address space is termed "system space." All processes use the same address translation map for system space, so system space is shared among all processes. The address map for system space is not context switched.

5.2.3 Virtual Address Format

The VAX-11 processor generates a 32-bit virtual address for each instruction and operand in memory. As the process executes, the system translates each virtual address to a physical address. The virtual address has the following format:

```
3 1
+-----------------+------------------+
| 9 8             | byte #            |
| VPN             |                  |
+-----------------+------------------+
```

Figure 5-2
Virtual Address Format

VPN <31:9> The Virtual Page Number field specifies the virtual page to be referenced. The virtual address space contains 8,388,608 (2**23) pages of 512 bytes each.

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

When bit 31 is one, the address is in the system space. When bit 31 is zero, the address is in the per-process space.

Within the per-process space, bit 30 distinguishes between the program and control regions. When bit 30 is one, the control region is referenced, and when it is zero, the program region is referenced.
5.2.4 Virtual Address Space Layout

The layout of virtual address space is illustrated in Figure 5-1. Note that access to each of the three regions (P0, P1, System) is controlled by a length register (P0LR, PILR, SLR). Within the limits set by the length registers, the access is further controlled by page tables that specify the validity, access requirements, and physical location of each page in the memory.

5.3 MEMORY MANAGEMENT CONTROL

The action of translating a virtual address to a physical address is governed by the setting of the Memory Mapping Enable (MME) bit in the MAPEN internal processor register. Figure 5-3 illustrates the privileged MAP ENable register.

```
3
1
0
```

Figure 5-3
MAP ENable Register (MAPEN)
(to read: MFPR #56, dst.wl)
(to write: MTFR src.rl, #56)

MAPEN<0> is the Memory Mapping Enable (MME) bit. When MME is set to 1, memory management is enabled. When MME is set to 0, memory management is disabled. At processor initialization time, MAPEN is initialized to 0.

5.3.1 Memory Management Disabled

Setting MME to 0 turns off address translation and access control. Virtual address bit n, VA<n>, is copied directly to the corresponding physical address bit, PA<n>, for n = 0 to 29. VA<31:30> are ignored; PA<31:30> are always zero. VA<n> is ignored if PA<n> doesn't exist. (The number of PA bits is implementation dependent.)

```
PA = VA<29:0> modulo (2** number of PA bits)
```

There is no page protection: all accesses are allowed in all modes. No modify bit is maintained.
5.4 ADDRESS TRANSLATION

When MME is a 1, address translation and access control are on. The processor uses the following to determine whether an intended access is allowed:

1. The virtual address, which is used to index a page table,
2. The intended access type (read or write), and
3. The current privilege level from the Processor Status Longword, or Kernel level for page table mapping references.

If the access is allowed and the address can be mapped, the result is the physical address corresponding to the specified virtual address.

The intended access is READ if the operation to be performed is a read. The intended access is WRITE if the operation to be performed is a write. If the operation to be performed is a modify (that is, read followed by write) the intended access is specified as a WRITE.

If an operand is an address operand, then no reference is made. Hence the page need not be accessible and need not even exist.

5.4.1 Page Table Entry (PTE)

The CPU uses a Page Table Entry (PTE) to translate virtual addresses to physical addresses. Figure 5-4a illustrates the PTE format.

```
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+
|        |        |        |        |        |        |        |        |        |        |
|        |        |        |        |        |        |        |        |        |        |
| V       | PROT    | M | Z | OWN | S | S | PPN |        |        |
| +--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+
```

Figure 5-4a
Page Table Entry

- **V** *<31>*: Valid bit - governs the validity of the M bit and PPN field. V=1 for valid; V=0 for not valid. When V=0, the M and PPN fields are reserved for DIGITAL software.

- **PROT** *<30:27>*: PROTection field - this field is always valid and is used by the CPU hardware even when V=0.

- **M** *<26>*: Modify bit - When the Valid bit is clear, M is not used by CPU hardware, and is reserved for DIGITAL software and I/O devices. When the Valid bit is set, M shows whether the page has been modified. If M is clear, the page has not
been modified. If M is set, the page may have been modified.

M is cleared only by software. It is set by CPU hardware on a successful write or modify to the page. In addition, it may be set by the probe-write instruction (PROBEW) or by an implied probe-write. M is not set if the page is inaccessible. Beyond that, it is UNPREDICTABLE whether M is set if a fault occurs in an instruction which would otherwise have modified the page.

For example, if a write reference crosses a page boundary where the first page is not accessible and the second page is accessible, the reference will fault. M is unchanged in the PTE mapping the first page. It is UNPREDICTABLE whether M is set in the PTE mapping the second page.

It is UNPREDICTABLE whether the modification of a process PTE<\( M \) bit causes modification of the system PTE that maps that process page table. Note that the update of the M bit is not interlocked in a multiprocessor system.

OWN <24:23> OWNer bits - reserved for DIGITAL software use as the access mode of the owner of the page (that is, the mode allowed to alter the page protection or to delete the page); not examined or altered by any hardware.

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

Z <25> Zero bit - bit 25 is RESERVED to DIGITAL and must be zero. The hardware does not necessarily test that this bit is zero because the PTE is established by privileged software.

S <22:21> Software bits - bits 22, and 21 are reserved for DIGITAL software.

(Software symbols defined for the above fields use PTE$ as the prefix.)

The operating system software uses some combinations of the software bits to implement its page management data structures and functions. Among the functions implemented this way are initialize-pages-with-zeros, copy-on-reference, page sharing, and transitions between active and swapped-out states. VAX/VMS encodes these functions in PTEs whose Valid bit, PTE<\( 31 \)>, is a 0 and processes them whenever a page fault occurs.
5.4.2 Page Table Entry (PTE) For I/O Devices

Some I/O devices, such as the DR32, use VAX-11 memory management to translate addresses. These I/O devices use a Page Table Entry format which is an extension of that in Figure 5-4a used by the CPU. The extended PTE implements for I/O hardware some functions that the CPU does with software using software bits and page faults. In particular, PTE bits 31, 26, and 22 are decoded into four combinations. Some of these are used in the same way as in the CPU PTE format, and some are used in different ways. The four combinations are:

- **PTE<31,26,22>** PTE Type
  
  1  x  x  Valid PFN
  0  0  0  Valid PFN
  0  0  1  Global Page Table Index
  0  1  x  Invalid, I/O abort

and their interpretations are:

**PTE<31,26,22>=1xx**, Figure 5-4b. PTE<20:0> is a valid PFN field. This is identical to the PFN field illustrated in Figure 5-4a for the CPU PTE.

```
  3 3  2 2 2 2 2 2 2 2 2 2 2
  1 0  7 5 5 4 3 2 1 0
  +-------------------------------+ 0
  |1| PROT |M|Z|OWN|S|S| |PFN
  +-------------------------------+
```

**Figure 5-4b**

**PTE<31,26,22>=000**, Figure 5-4c. PTE<20:0> is a valid PFN field. This is identical to the PFN field illustrated in Figure 5-4a for the CPU PTE.

```
  3 3  2 2 2 2 2 2 2 2 2 2 2 2 2 2
  1 0  7 5 5 4 3 2 1 0
  +-------------------------------+ 0
  |0| PROT |0|Z|OWN|0|S| |PFN
  +-------------------------------+
```

**Figure 5-4c**

**PTE<31,26,22>=001**, Figure 5-4d. PTE<21:0> is a Global Page Table Index (GPTX). The I/O device has a Global page table Base Register (GBR) which is loaded by software with a system virtual address. The I/O device calculates GBR + GPTX * 4 to get the system virtual address of a second PTE. The second PTE must contain a valid PFN, and must have PTE<31,26,22> equal to either 000 or 1xx, binary. If either of these requirements is not met, the result is UNDEFINED. For those devices that use it, the PROTECTION field always comes from the first PTE.
Figure 5-4d
PTE<31,25,22>=001, Global Page Table Index

PTE<31,25,22>=01x, Figure 5-4e. This PTE format is RESERVED to DIGITAL. I/O devices will abort in a DEVICE DEPENDENT manner.

Figure 5-4e
PTE<31,25,22>=01x, Invalid, I/O abort

I/O devices may look at and check the PROTection field or modify the M bit; this is DEVICE DEPENDENT. Those devices that do use them, use them the same way the CPU does.

I/O devices that do memory mapping use the same SPT as the CPU, but they have their own copies of the SBR and SLR. Buffer addresses are described in terms of a system virtual address of the PTE for the first buffer page and a byte offset within that page. In addition the I/O devices use a Global Page Table in memory and an I/O hardware Global page table Base Register (GTR) which must be loaded by software.

5.4.3 Changes To Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, VMS sets and clears the valid bit and changes the PFN field as pages are swapped in and out.

The software must guarantee that each PTE is always consistent within itself. Changing a PTE one field at a time may give incorrect system operation. An example would be to set PTE<V> with one instruction before establishing PTE<PFN> with another. An interrupt routine between the two instructions could use an address that would map using the inconsistent PTE. The software can solve this problem by building a new PTE in a register and then moving the new PTE to the page table with a single instruction such as MOVFL.

Multiprocessing makes the problem more complicated. Another processor, be it another CPU or an I/O processor, can reference the same page tables that the first CPU is changing. The second processor must always read consistent PTEs. In order to guarantee this, first note that PTEs
are longwords, longword-aligned. Then two requirements must be met:

1. Whenever the software modifies a PTE in more than one byte, it must use a longword, longword-aligned, write-destination instruction, such as `MOVL`, and

2. The hardware must guarantee that a longword, longword-aligned write is an "atomic" operation. That is, a second processor cannot read (or write over) any of the first processor's partial results.

### 5.5 ACCESS CONTROL

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

#### 5.5.1 Processor Modes

In the order of most privileged to least privileged, the four processor modes 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 access mode of a running process is the current processor mode, stored in the Current Mode field of the Processor Status Longword (PSL) (see the Chapter on Exceptions and Interrupts).

#### 5.5.2 Protection Code

Every page in the virtual address space is protected according to its use. Even though all of the system space is shared, in that the program may generate any address, the program may be prevented from modifying, or even accessing portions of it. A program may also be prevented from accessing or modifying portions of per-process space.
For example, in system space, scheduling queues are highly protected, whereas library routines may be executable by code of any privilege. Similarly per-process accounting information may be in per-process space, but highly protected, while normal user code in per-process spaces is executable at low privilege.

Associated with each page is a protection code that describes the accessibility of the page for each processor mode. The code allows a choice of protection for each processor mode, within the following limits:

1. Each level's access can be read-write, read-only, or no-access.

2. If any level has read access then all more privileged levels also have read access.

3. If any level has write access then all more privileged levels also have write access.

The protection codes for the 15 combinations of page protection are encoded in a 4 bit field in the Page Table Entry as follows:
<table>
<thead>
<tr>
<th>CODE</th>
<th>MNEUMONIC</th>
<th>PRIVILEGE LEVEL</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>K E S U</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>NA - - -</td>
<td>no ACCESS</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>- - -</td>
<td>RESERVED</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>KW RW - -</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>KR R - -</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>UW RW RW RW</td>
<td>ALL ACCESS</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>EW RW RW -</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>ERKW RW R -</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>ER R R -</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>SW RW RW -</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>SREW RW RW R</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>SRKW RW R R</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>SR R R R</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>URSW RW RW R</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>UREW RW R R</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>URKW RW R R</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>UR R R R</td>
<td></td>
</tr>
</tbody>
</table>

**Key**

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

**Figure 5-5**
Protection Mnemonics

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

This encoding was chosen to simplify hardware access checking for implementations not using a table decoder. The access is allowed if:

\[
\text{CODE NEQU 0} \text{ AND }
\left\{ \text{CODE EQUU 4 OR (CM LSSU WM) OR (READ AND (CM LEQU RM))} \right\}
\]

CM is current processor mode
RM is left 2 bits of code
WM is one's complement of right 2 bits of code
5.5.3 Length Violation

Every valid virtual address lies within bounds determined by the addressing region (P0, P1, or System) and the associated length register (P0LR, P1LR, or SLR). Virtual addresses outside these bounds cause a length violation. The addressing bounds algorithm is a simple limit check whose formal notation is:

\[
\text{case VAddr}\langle31:30\rangle
\]

\[
\text{set }
\]

\[
[0]: \quad \text{IP0 region}
\]

\[
\text{if ZEXT( VAddr}\langle29:9\rangle \) GEQU P0LR}
\]

\[
\text{then \{length violation\};}
\]

\[
[1]: \quad \text{IP1 region}
\]

\[
\text{if ZEXT( VAddr}\langle29:9\rangle \) LSSU P1LR}
\]

\[
\text{then \{length violation\};}
\]

\[
[2]: \quad \text{IS region}
\]

\[
\text{if ZEXT( VAddr}\langle29:9\rangle \) GEQU SLR}
\]

\[
\text{then \{length violation\};}
\]

\[
[3]: \quad \text{reserved region}
\]

\[
\text{tes;}
\]

5.5.4 Access Control Violation Fault

An access control fault occurs if an illegal access is attempted, as determined by the current PSL mode and the page's protection field, or if the address causes a length violation.

5.5.5 Access Across A Page Boundary

If an access is made across a page boundary, the order in which the pages are accessed is UNPREDICTABLE. However, for a given page, access control violation always takes precedence over translation not valid.

5.5.6 System Space Address Translation

A virtual address with \langle31:30\rangle=2 is an address in the system virtual address space.
Memory Management
ACCESS CONTROL

3 3 2 9 8 0
+-----------------+ Byte # |
| 2 | --------System Virtual Page No. (SVPN)------>
+-----------------+ | |

Figure 5-6
System Virtual Page Format

The system virtual address space is defined by the System Page Table (SPT), which is a vector of Page Table Entries (PTEs). The SPT is always located in physical address space. The base address of the SPT is also a physical address and is contained in the System Base Register (SBR). The size of the SPT in longwords (that is, the number of PTEs) is contained in the System Length Register (SLR). The SBR points to the first PTE in the SPT. In turn, this PTE maps the first page of System Space, that is, virtual byte address 80000000(hex).

The PTEs in the System Page Table contain the mapping information themselves, or point to the mapping information in the Global Page Table if the PTE is in GPTX format. (See the section on PTEs for I/O devices for a description of the GPTX format.)

3 3 2 9 8 0
+-----------------+ Physical Longword Address |
| MBZ | | |
+-----------------+ |

Figure 5-7
System Base Register (SBR)
(to read: MFPR #12, dst.w1)
(to write: MTPR src.r1, #12)

3 2 2 1 0
+-----------------+ Length of SPT in longwords |
| MBZ | |
+-----------------+ |

Figure 5-8
System Length Register (SLR)
(to read: MFPR #13, dst.w1)
(to write: MTPR src.r1, #13)

Bits <31:9> of the virtual address contain the Virtual Page Number. 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 the section at the end of this chapter on Sharing.) The length field in the System Length Register requires 22 bits to express the values 0 through 2**21 inclusive. At processor initialization time, the contents of both registers are UNPREDICTABLE. Figure 5-9 illustrates the
translation of a system virtual address to a physical address.

Figure 5-9
System Virtual to Physical Translation

The algorithm to generate a physical address from a system region virtual address is:

\[ \text{SYS\_PA} = (\text{SBR} + 4 \times \text{SVA} < 29:9 > < 20:0 > ') \text{SVA} < 8:0 > \]

!System Region
Note

For all occurrences within this chapter, the parentheses indicate "contents of," the angle brackets indicate referenced bits, and the apostrophe indicates concatenation.

5.5.7 Process Space Address Translation

The process virtual address space is divided into two equal sized, separately mapped regions. If virtual address bit 30 is 0, the address is in region P0. If virtual address bit 30 is 1, the address is in region P1.

The P0 region maps a virtually contiguous area that begins at the smallest address (0) in the process virtual space and grows in the direction of larger addresses.

P0 is typically used for program images and can grow dynamically.

The P1 region maps a virtually contiguous area that begins at the largest address (\(2^{31} - 1\)) in the process virtual space and grows in the direction of smaller addresses.

P1 is typically used for system-maintained, per-process context. It may grow dynamically for the user stack.

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

There is a significant reason to address process page tables in virtual rather than physical space. A physically addressed process page table that required more than a page of PTEs (that is, that mapped more than 64K bytes of process virtual space) would require physically contiguous pages. Such a requirement would make dynamic allocation of process page table space very awkward since a running system tends to fragment storage into page-sized areas.

A process space address translation that causes a translation buffer miss will cause one memory reference for the process PTE. If the virtual address of the page containing the process PTE is also missing from the translation buffer, a second memory reference is required.
When a process Page Table Entry is fetched, a reference is made to System Space. This reference is made as a kernel read. Thus the system page containing a process page table is either "No Access" (that is, protection code zero) or will be accessible (protection code non-zero). Similarly, a check is made against the System page table Length Register (SLR). Thus, the fetch of an entry from a process page table can result in access or length violation faults (see the section on Faults and Parameters).

5.5.8 P0 Region

The P0 region of the address space is mapped by the P0 Page Table (P0PT) which is defined by the P0 Base Register (P0BR) and the P0 Length Register (P0LR). The P0BR contains a virtual address in the system region which is the base address of the P0 Page Table. Figure 5-10 illustrates the P0 Base Register. The P0LR contains the size of the P0PT in longwords, that is, the number of Page Table Entries. Figure 5-11 illustrates the P0 Length Register. The Page Table Entry addressed by the P0 Base Register maps the first page of the P0 region of the virtual address space, that is, virtual byte address 0.

The PTEs in the P0 Page Table contain the mapping information themselves, or point to the mapping information in the Global Page Table if the PTE is in GPTX format. (See the section on PTEs for I/O devices for a description of the GPTX format.)

```
  3 3 2
  1 0 9
+---+------------------------------+---+
| 2 | System Virtual Longword Address |MBZ|
+---+------------------------------+---+

Figure 5-10
P0 Base Register (P0BR)
(to read: MFPR #8, dst.w1)
(to write: MTPR src.r1, #8)

  3  2 2 2 2 2 2
  1  7 6 4 3 2 1
+---+---+---+---+---+
|MBZ|IGN|MBZ|
+---+---+---+---+

Figure 5-11
P0 Length Register (P0LR)
(to read: MFPR #9, dst.w1)
(to write: MTPR src.r1, #9)
```

The Virtual Page Number is contained in bits <29:9> of the virtual address. A 22-bit length field is required to express the values 0 through 2**21 inclusive. There could be as many as 2**21 pages in the P0 region.
POLR<25:24> are ignored on MTPR and read back 0 on MFPR. At processor initialization time, the contents of both registers are UNPREDICTABLE. An attempt to load POBR with a value less than 2**31 or greater than 2**31 + 2**30 - 4 results in a reserved operand fault in some implementations. Figure 5-12 illustrates the PO virtual address to physical address translation.
The algorithm to generate a physical address from a P0 region virtual address is:

\[
\begin{align*}
\text{PVA}_{\text{PTE}} & = \text{P0BR} + 4 \times \text{PVA}_{<29:9>} \\
\text{PTE}_{\text{PA}} & = (\text{SBR} + 4 \times \text{PVA}_{\text{PTE},<29:9>})_{<20:0>} \times \text{PVA}_{<8:0>} \\
\text{PROC}_{\text{PA}} & = (\text{PTE}_{\text{PA}})_{<20:0>} \times \text{PVA}_{<8:0>}
\end{align*}
\]
5.5.9 P1 Region

The P1 region of the address space is mapped by the P1 Page Table (P1PT) which is defined by the P1 Base Register (P1BR) and the P1 Length Register (P1LR). Because P1 space grows towards smaller addresses, and because a consistent hardware interpretation of the base and length registers is desirable, P1BR and P1LR describe the portion of P1 space that is NOT accessible. Figure 5-13 illustrates the P1 Base Register. Figure 5-14 illustrates the P1 Length Register. Note that P1LR contains the number of nonexistent PTEs. P1BR contains a virtual address of what would be the PTE for the first page of P1, that is, virtual byte address 40000000(hex).

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

The PTEs in the P1 Page Table contain the mapping information, or point to the mapping information in the Global Page Table if the PTE is in GPTX format. (See the section on PTEs for I/O devices for a description of the GPTX format.)

```
  3             2 1 0
  1
+-------------------------+---------+--------+
|                         | Virtual Longword Address | MBZ |
+-------------------------+---------+--------+

Figure 5-13
P1 Base Register (P1BR)
( to read: MFPR #10, dst.wl )
( to write: MTPR src.r1, #10 )

  3 3                2 2
  1 0                2 1
+-------------------+--------+
| MBZ               | 2**21  - Length of P1PT in longwords |
+-------------------+--------+

Figure 5-14
P1 Length Register (P1LR)
( to read: MFPR #11, dst.wl )
( to write: MTPR src.r1, #11 )
```

P1LR<31> is ignored on MTPR and reads back 0 on MFPR. At processor initialization time, the contents of both registers are UNPREDICTABLE. An attempt to load P1BR with a value less than 2**31 - 2**23 (7F800000, hex) or greater than 2**31 + 2**30 - 2**23 - 4 results in a reserved operand fault in some implementations.
The algorithm to generate a physical address from a Pl region virtual address is:

\[ \text{PVA} \_ \text{_PTE} = \text{PlBR} + 4 \times \text{PVA} <29:9> \]
\[ \text{PTE} \_ \text{PA} = (\text{SBR} + 4 \times \text{PVA} \_ \text{PTE} <29:9>) <20:0> \times \text{PVA} <8:0> \]
\[ \text{PROC} \_ \text{PA} = (\text{PTE} \_ \text{PA}) <20:0> \times \text{PVA} <8:0> \]
5.6 TRANSLATION BUFFER

In order to save actual memory references when repeatedly referencing the same pages, a hardware implementation may include a mechanism to remember successful virtual address translations and page states. Such a mechanism is termed a translation buffer.

When the process context is loaded with LDFCTX, the translation buffer is automatically updated (that is, the process virtual address translations are invalidated). However, when the software changes any part of a valid Page Table Entry for the system or a current process region, it must also move a virtual address within the corresponding page to the Translation Buffer Invalidate Single (TBIS) register with the MTPR instruction. Figure 5-16 illustrates the TBIS register.

Additionally, when the software changes a System Page Table Entry which maps any part of the current process page table, all process pages so mapped must be invalidated in the translation buffer. They may be invalidated by moving an address within each such page into the TBIS register. They may also be invalidated by clearing the entire translation buffer. This is done by moving 0 to the Translation Buffer Invalidate All (TBIA) register with the MTPR instruction. Figure 5-17 illustrates the TBIA register.

The translation buffer must not store invalid PTEs. Therefore, the software is not required to invalidate translation buffer entries when making changes for PTEs that are already invalid.

When the location or size of the system map is changed (SBR, SLR) the entire translation buffer must be cleared.

Whenever Memory Management Enable (MME) is a 0, the contents of the translation buffer are UNPREDICTABLE. Therefore, before enabling memory management at processor initialization time, or any other time, the entire translation buffer must be cleared.

---

![Figure 5-16](attachment:image)

Translation Buffer Invalidate Single (TBIS)
(to write: MTPR src rl, #58)
An internal processor register is available for interrogating the presence of a valid translation in the translation buffer. When a virtual address is written to the TBCHK register with a MTPR instruction, the condition code V bit is set if the translation buffer holds a valid translation for that virtual page. The specification of the TBCHK register is based on VAX/VMS usage. The TBCHK register is reserved for Digital use. Its specification is subject to change without prior notice.

### 5.7 FAULTS AND PARAMETERS

Two types of faults are associated with memory mapping and protection (see the chapter on Exceptions and Interrupts for a description of faults). A Translation Not Valid Fault is taken when a read or write reference is attempted through an invalid PTE (PTE<31>=0). An Access Control Violation Fault is taken when the protection field of the PTE indicates that the intended page reference in the specified access mode would be illegal. Note that these two faults have distinct vectors in the System Control Block. If both faults could occur, then the Access Control Violation Fault takes precedence. An Access Control Violation Fault is also taken if the virtual address referenced is beyond the end of the associated page table. Such a "length violation" is essentially the same as referencing a PTE that specifies "No Access" in its protection field. To avoid having the fault software recompute the length check, a "length violation" indication is stored in the fault parameter word illustrated in Figure 5-18.

![Figure 5-17](attachment:translation_buffer_invalidate_all_tbia.png)
Translation Buffer Invalidate All (TBIA)
(to write: MTPR src.r1, #57)

![Figure 5-18](attachment:fault_parameter_block.png)
Fault Parameter Block
The same parameters are stored for both types of fault. The first parameter pushed on the stack after the PSL and PC is some virtual address in the same page with the virtual address that caused the fault. A Process Space reference can result in a System Space virtual reference for the PTE. If the PTE reference faults, the virtual address that is saved is the process virtual address. In addition, a 1 is stored in bit 1 of the fault parameter word if the fault occurred in the per-process PTE reference.

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

- **L**<sup>0</sup>  
  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**<sup>1</sup>  
  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**<sup>2</sup>  
  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.

### 5.8 PRIVILEGED SERVICES AND ARGUMENT VALIDATION

#### 5.8.1 Changing Access Modes

Four instructions allow a program to change its access mode to a more privileged mode and transfer control to a service dispatcher for the new mode.

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

These instructions, described in detail in the chapter on Exceptions and Interrupts, provide the normal mechanism for less privileged code to call more privileged code. When the mode transition takes place, the previous mode is saved in the Previous Mode field of the PSL, thus allowing the more privileged code to determine the privilege of its caller.
5.8.2 Validating Address Arguments (PROBE instructions)

Two instructions, PROBER and PROBEW, allow privileged services to check addresses passed as parameters. To avoid protection holes in the system, a service routine must always verify that its less privileged caller could have directly referenced the addresses passed as parameters (see the appendix on Address Validation Rules). The PROBE instructions do this verification.
PROBEX PROBE ACCESSIBILITY

Purpose: verify that arguments can be accessed

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

Operation:

\[
\text{probe\_mode} <- \text{MAXU (mode<1:0>, PSL<PRV\_MOD>)}
\]

\[
\text{condition\_codes} <- \{\text{accessibility of base}\} \text{ and }
\{\text{accessibility of } \text{base+ZEXT(len)-1}\}
\text{ using probe\_mode}
\]

Condition Codes:

\[
N \leftarrow 0;
Z \leftarrow \text{if \{both\ accessible\} then } 0 \text{ else } 1;
V \leftarrow 0;
C \leftarrow C;
\]

Exceptions:

translation not valid

Opcodes:

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

Description:

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

The protection is checked against the larger (and therefore less privileged) of the modes specified in bits <1:0> of the mode operand and 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<previous-mode>.

Example:

\[
\text{MOVL 4(AP),R0} \quad ;\text{Copy the address of first arg so that }
\text{it can't be changed.}
\text{PROBER #0,#4,(R0)} \quad ;\text{Verify that the longword pointed to by}
\text{the first arg could be read by the }
\text{previous access mode.}
\]
;Note that the arg list itself must
;already have been probed
;Branch if either byte gives an access
;violation.
;Copy length and address of buffer args
;so that they can't change.
;Verify that the buffer described by the
;2nd and 3rd args could be written by
;the previous access mode.
;Note that the arg list must already
;have been probed and that the 2nd arg
;must be known to be less than 512.
;Branch if either byte gives an access
;violation.

Flows:

The following flows describe the operation of PROBE on each of the virtual addresses it is checking. Note that probing an address returns only the accessibility of the page(s) and has no effect on their residency. However, probing a process address may cause a page fault in the system address space on the per-process page tables.

1. Look up the virtual address in the translation buffer. If found, use the associated protection field to determine the accessibility and EXIT.

2. Check for length violation for System or per-Process address as appropriate. See elsewhere in this chapter for the length violation check flows. If length violation then return No Access and EXIT.

3. If System virtual address, form physical address of PTE, fetch the PTE, use the protection field to determine the accessibility and EXIT.

4. For per-Process virtual address, must do a virtual memory reference for the PTE.
   1. Look up the virtual address of the PTE in the translation buffer, form the physical address of the PTE if found, fetch the PTE, use the protection field to determine the accessibility and EXIT.
   2. Check the System virtual address of the PTE for length violation. If length violation, then return No Access and EXIT.
   3. Tl <- Page Table Entry for the page containing the per-process PTE.
   4. If the protection field of Tl indicates no access (not even readable by Kernel), then return No Access and EXIT. A no access, not valid pointer to a page of PTE's conserves
storage space for a page full of no access, not valid PTE's.

5. If the valid bit in T1 is 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 PPN field of T1 (see the section on System Space Address Translation), fetch the PTE, use the protection field to determine the accessibility, and EXIT.

5.8.3 Notes On The PROBE instructions

1. If the Valid bit of the examined Page Table Entry is set, it is UNPREDICTABLE whether the Modify bit of the examined Page Table Entry is set by a PROBEW. If the Valid bit is clear, the Modify bit is not changed.

2. Except for 1, above, the valid bit of the Page Table Entry, PTE<31>, mapping the probed address is ignored.

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

4. On the probe of a process virtual address, if the valid bit of the system Page Table Entry is 0 then a Translation Not Valid Fault occurs. This allows for the demand paging of the process page tables.

5. On the probe of a process virtual address, if the protection field of the system Page Table Entry indicates No Access, then a status of "not-accessible" is given. Thus, a single No Access Page Table Entry in the system map is equivalent to 128 No Access Page Table Entries in the process map.
CHAPTER 6
EXCEPTIONS AND INTERRUPTS

12-Dec-80 -- Rev 7.1

6.1 INTRODUCTION

At certain times during the operation of a system, events within the system require the execution of particular pieces of software outside the explicit flow of control. The processor transfers control by forcing a change in the flow of control from that explicitly indicated in the currently executing process.

Some of the events are relevant primarily to the currently executing process, and normally invoke software in the context of the current process. The notification of such events is termed an exception.

Other events are primarily relevant to other processes, or to the system as a whole, and are therefore serviced in a system-wide context. The notification process for these events is termed an interrupt, and the system-wide context is described as "executing on the interrupt stack" (IS). Further, some interrupts are of such urgency that they require high-priority service, while others must be synchronized with independent events. To meet these needs, the processor has priority logic that grants interrupt service to the highest priority event at any point in time. The priority associated with an interrupt is termed its interrupt priority level (IPL).
6.1.1 Processor Interrupt Priority Levels (IPL)

The processor has 31 interrupt priority levels (IPL), divided into 15 software levels (numbered, in hex, 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 on software generated interrupts later in this chapter). Once a software interrupt request is made, it will be cleared by the hardware when the interrupt is taken.

Interrupt levels 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 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 (IF, 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 BISPW and BICPSW instructions described in Chapter 4.

A fault is an exception condition that occurs during an instruction, and that leaves the registers and memory in a consistent state such that elimination of the fault condition and restarting the instruction will give correct results. Note that faults do not always leave everything as it was prior to the faulted instruction, they only restore enough to allow restarting. Thus, the state of a process that faults may not be the same as that of a process that was interrupted at the same point.

An abort is an exception condition that occurs during an instruction, and potentially leaves the registers and memory indeterminate, such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone.

6.1.4 Contrast Between Exceptions And Interrupts

Generally exceptions and interrupts are very similar. When either is initiated, both the processor status (PSL) and the program counter (PC) are pushed onto the stack. However there are seven important differences:

1. An exception condition is caused by the execution of the current instruction while an interrupt is caused by some activity in the computing system that may be independent of the current instruction.

2. An exception condition is usually serviced in the context of the process that produced the exception condition, while an interrupt is serviced independently from the currently running process.

3. The IPL of the processor is usually not changed when the processor initiates an exception, while the IPL is always raised when an interrupt is initiated.
4. Exception service routines usually execute on a per-process stack while interrupt service routines normally execute on a per-CPU stack.

5. Enabled exceptions are always initiated immediately no matter what the processor IPL is, while interrupts are held off until the processor IPL drops below the IPL of the requesting interrupt.

6. Most exceptions can not be disabled. However, if an exception causing event occurs while that exception is disabled, no exception is initiated for that event even when enabled subsequently. This includes overflow which is the only exception whose occurrence is indicated by a condition code (V). If an interrupt condition occurs while it is disabled, or the processor is at the same or higher IPL, the condition will eventually initiate an interrupt when the proper enabling conditions are met if the condition is still present.

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

When an exception or an interrupt is serviced, the processor status must be preserved so that the interrupted process may continue normally. Basically, this is done by automatically saving the Program Counter (PC) and the Processor Status Longword (PSL). These are later restored with the Return from Exception or Interrupt instruction (REI). Any other status required to correctly resume an interruptable instruction is stored in the general registers. Process context such as the mapping information is not saved or restored on each interrupt or exception. Instead, it is saved and restored only when process context switching is performed. Refer to the LDPCTX and SVPCTX instructions in chapter 7. Other processor status is changed even less frequently; refer to the privileged register descriptions in chapter 9.

The Processor Status Longword (PSL) is a longword consisting of a word of privileged processor status concatenated with the Processor Status Word (PSW). Refer to chapter 2 for a description of the PSW. The PSL is automatically saved on the stack when an exception or interrupt occurs and is saved in the PCB on a process context switch. The PSL can also be stored by the MOVPSL instruction; refer to chapter 4. (The terms current PSL and saved PSL are used to distinguish between this status information when it is in the processor and when copies of it are materialized in memory.)

Bits <31:21> of the current PSL can be changed explicitly only by executing a return from exception or interrupt instruction (REI). REI considers the current mode when restoring the PSL, and faults if a program attempts to increase its privilege by this means. Thus REI is available to all software including user exception handling routines.

```
 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1
1 0 9 8 7 6 5 4 3 2 1 0 6 5 8 7 6 5 4 3 2 1 0

+--------+--------+--------+--------+--------+--------+--------+--------+
|C|T|   |F|I|CUR|PRV|M|   |D|F|I|T|N|Z|V|C|
|M|P|MBZ|P|S|MOD|MOD|B| IPL | MBZ |V|U|V|   |   |   |   |
|   |   |   |D|   |   |   |Z|   |   |   |   |   |   |   |
+--------+--------+--------+--------+--------+--------+--------+--------+
                  /                        \  
                  \                      --PSW-----
```

Processor Status Longword

At bootstrap time, PSL is cleared except for IPL and IS.
### Bits Description

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>Condition Codes: N, Z, V, C (See chapter 2)</td>
</tr>
<tr>
<td>4</td>
<td>Trace enable (T). When set at the beginning of an instruction, causes TP to be set. When TP is set at the end of an instruction, a trace fault is taken before the execution of the next instruction. When TP is clear, no trace exception occurs. Most programs should treat T as UNPREDICTABLE because it is set by debuggers and trace programs for tracing and for proceeding from a breakpoint.</td>
</tr>
<tr>
<td>5</td>
<td>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.)</td>
</tr>
<tr>
<td>6</td>
<td>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.</td>
</tr>
<tr>
<td>7</td>
<td>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.)</td>
</tr>
<tr>
<td>15:8</td>
<td>Reserved to DIGITAL, must be zero.</td>
</tr>
<tr>
<td>20:16</td>
<td>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).</td>
</tr>
<tr>
<td>21</td>
<td>Reserved to DIGITAL, must be zero.</td>
</tr>
<tr>
<td>22:23</td>
<td>Previous Access Mode (PRV_MOD). Loaded from current mode by exceptions and CHMx instructions, cleared by interrupts, and restored by REI.</td>
</tr>
<tr>
<td>25:24</td>
<td>Current Access Mode (CUR_MOD). The access mode of the currently executing process, as follows:</td>
</tr>
<tr>
<td></td>
<td>0 - KERNEL</td>
</tr>
<tr>
<td></td>
<td>1 - EXECUTIVE</td>
</tr>
<tr>
<td></td>
<td>2 - SUPERVISOR</td>
</tr>
<tr>
<td></td>
<td>3 - USER</td>
</tr>
</tbody>
</table>
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.

First Part Done (FPD). When set, execution of the instruction addressed by PC cannot simply be started at the beginning, and must be restarted at some other, implementation specific, point in its operation. If FPD is set and the exception or interrupt service routine modifies FPD, the general registers, or the saved PSL (except for T or TP), the results of the restarted instruction's execution are UNPREDICTABLE. If a routine sets FPD, the results are also UNPREDICTABLE. However, if software is simulating unimplemented instructions, it may make free use of FPD in its simulation. If the hardware encounters a reserved instruction with FPD set, a reserved instruction fault is taken with the saved PSL<FPD> set.

Reserved to DIGITAL, must be zero.

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.

Compatibility Mode (CM). When set the processor is in PDP-11 compatibility mode (see chapter 10). When CM is clear, the processor is in native mode.
6.3 INTERRUPTS

The processor services interrupt requests between instructions. The processor also services interrupt requests at well defined points during the execution of long, iterative instructions such as the string instructions. For these instructions, in order to avoid saving additional instruction state in memory, interrupts are initiated when the instruction state can be completely contained in the registers, PSL, and PC.

The following events cause interrupts:

1. Device completion (IPL 10-17 hex)
2. Device error (IPL 10-17 hex)
3. Device alert (IPL 10-17 hex)
4. Device memory error (IPL 10-17 hex)
5. Console terminal transmit and receive (IPL 14 hex)
6. Interval timer (IPL 18 hex)
7. Recovered memory or bus or processor errors (implementation specific, IPL 18 to 1D hex); The VAX-11/780 processor interrupts at 1B on memory errors.
8. Unrecovered memory or bus or processor errors (implementation specific, IPL 18 to 1D hex)
9. Power fail (IPL 1E hex)
10. Software interrupt invoked by MTPR #SIRR (IPL 01 to 0F hex)
11. AST delivery when REI restores a PSL with mode greater than or equal to ASTLVL (see chapter 7) (IPL 02)

Each device controller has a separate set of interrupt vector locations in the system control block (SCB). Thus interrupt service routines do not need to poll controllers in order to determine which controller interrupted. The vector address for each controller is fixed by hardware.

In order to reduce interrupt overhead, no memory mapping information is changed when an interrupt occurs. Thus the instructions, data, and contents of the interrupt vector for an interrupt service routine must be in the system address space or present in every process at the same address.
6.3.1 Urgent Interrupts -- Levels 18-1F (Hex)

The processor provides 8 priority levels for use by urgent conditions including serious errors (e.g., machine check) and power fail. Interrupts on these levels are initiated by the processor upon detection of certain conditions. Some of these conditions are not interrupts. For example, Machine Check is usually an exception but it runs at a high priority level on the interrupt stack.

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

6.3.2 Device Interrupts -- Levels 10-17 (Hex)

The processor provides 8 priority levels for use by peripheral devices. Any given implementation may or may not implement all 8 levels of interrupts. The minimal implementation is levels 14-17 (hex) that correspond to the UNIBUS levels BR4 to BR7 if the system has a UNIBUS.
6.3.3 Software Generated Interrupts -- Levels 01-0F (Hex)

6.3.3.1 Software Interrupt Summary Register - The processor provides 15 priority interrupt levels for use by software. Pending software interrupts are recorded in the Software Interrupt Summary Register (SISR). The SISR contains 1's in the bit positions corresponding to levels on which software interrupts are pending. All such levels, of course, must be lower than the current processor IPL, or the processor would have taken the requested interrupt.

```
+----------------+----------------+-+
| 3              | 1 | 1                |
| 1              | 6 | 5                |
|                |   |                   |
+----------------+----------------+-+
+----------------+----------------+-+
|                  | Pending Software Interrupts | M |
|                  |                | B |
|                  | F E D C B A 9 8 7 6 5 4 3 2 1 | Z |
+----------------+----------------+-+
```

Software Interrupt Summary Register

The SISR is a read/write privileged register accessible only to privileged software (see Chapter 9). At bootstrap time, the contents of SISR is cleared. The mechanism for accessing it is:

MFPR #SISR,dst Reads the software interrupt summary register.

MTPR src,#SISR Loads it, but this is not the normal way of making software interrupt requests. It is useful for clearing the software interrupt system, and for reloading its state after a power fail, for example.

6.3.3.2 Software Interrupt Request Register - The software interrupt request register (SIRR) is a write-only four bit privileged register used for making software interrupt requests.

```
+----------------+----------------+----------------+----------------+----------------+
| 4 | 3 | 0               |
| 3 | 1 | ignored         |
+----------------+----------------+----------------+----------------+
```

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 MPPR 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.

```
+-----------------+-----------+-----------+
|                  |    5     |    4     |
|                  |-----------+-----------|
|                  | ignored; returns 0 |

Interrupt Priority Level Register
```

At bootstrap time, IPL is initialized to 31 (1F, hex).

Interrupt service routines must follow the discipline of not lowering IPL below their initial level. If they do, an interrupt at an intermediate level could cause the stack nesting to be improper. This would result in REI faulting. Actually, a service routine could lower the IPL if it ensures that no intermediate levels could interrupt, however this is probably unreliable code.
### 6.3.5 Interrupt Example

As an example, assume the processor is running in response to an interrupt at IPL5, it then sets IPL to 8, and then posts software requests at IPL3, IPL7, and IPL9. Then a device interrupt arrives at IPL11 (hex). Finally IPL is set back to IPL5. The sequence of execution is:

<table>
<thead>
<tr>
<th>event</th>
<th>state after event contents of IPL (hex)</th>
<th>SISR (hex)</th>
<th>IPL in PSL on stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(initial)</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #8,#IPL</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #3,#SIRR</td>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #7,#SIRR</td>
<td>8</td>
<td>88</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #9,#SIRR interrupts to device interrupts to</td>
<td>9</td>
<td>88</td>
<td>8,0</td>
</tr>
<tr>
<td>device interrupts to</td>
<td>11</td>
<td>88</td>
<td>9,8,0</td>
</tr>
<tr>
<td>device service routine REI</td>
<td>9</td>
<td>88</td>
<td>8,0</td>
</tr>
<tr>
<td>IPL9 service routine REI</td>
<td>8</td>
<td>88</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #5,#IPL changes IPL to 5 and the request for 7 is granted immediately</td>
<td>7</td>
<td>8</td>
<td>5,0</td>
</tr>
<tr>
<td>IPL7 service routine REI</td>
<td>5</td>
<td>88</td>
<td>0</td>
</tr>
<tr>
<td>initial IPL5 service routine REI back to IPL0 and the request for 3 is granted immediately</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IPL3 service routine REI</td>
<td>0</td>
<td>0</td>
<td>--</td>
</tr>
</tbody>
</table>
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 occurring as a consequence of an instruction
5. Tracing
6. Serious system failures
5.4.1 Arithmetic Traps/Faults

This section contains the descriptions of the exceptions that occur as the result of performing an arithmetic or conversion operation. They are mutually exclusive and all are assigned the same vector in the SCB, and hence the same signal "reason" code. Each of them indicates that an exception had occurred during the last instruction and that the instruction has been completed (trap) or backed up (fault). An appropriate distinguishing code is pushed on the stack as a longword:

```
+-------------------------------+ : (SP) |
| type code                     |
+-------------------------------+         |
| PC of next instruction to execute* |
|                                 |
| PSL                            |
```

*same as the instruction causing exception in case of fault

<table>
<thead>
<tr>
<th>type code (hex)</th>
<th>exception type</th>
<th>software mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>integer overflow</td>
<td>SRMSK_INT_OVF_T</td>
</tr>
<tr>
<td>2</td>
<td>integer divide by zero</td>
<td>SRMSK_INT_DIV_T</td>
</tr>
<tr>
<td>3</td>
<td>floating overflow</td>
<td>SRMSK FLT_OVF_T</td>
</tr>
<tr>
<td>4</td>
<td>floating/decimal divide by zero</td>
<td>SRMSKFLT_DIV_T</td>
</tr>
<tr>
<td>5</td>
<td>floating underflow</td>
<td>SRMSKFLT_UF_T</td>
</tr>
<tr>
<td>6</td>
<td>decimal overflow</td>
<td>SRMSK_DEC_OVF_T</td>
</tr>
<tr>
<td>7</td>
<td>subscript range</td>
<td>SRMSK_SUB_RNG_T</td>
</tr>
<tr>
<td>8</td>
<td>floating overflow</td>
<td>SRMSK FLT_OVF_F</td>
</tr>
<tr>
<td>9</td>
<td>floating divide by zero</td>
<td>SRMSK FLT_DIV_F</td>
</tr>
<tr>
<td>A</td>
<td>floating underflow</td>
<td>SRMSK FLT UND_F</td>
</tr>
</tbody>
</table>

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 (SRMSK_INT_OVF_T). Note that the instructions RET, REI, REMQUE, REMQHI, REMQTI, MOVTC, and BISP W 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, R0 through RG, 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 KFLTUND T).

6.4.1.6 Decimal String Overflow Trap - A decimal string overflow trap is an exception that indicates that the last instruction executed had a decimal string result too large for the destination string provided and that decimal overflow was enabled (DV set). The V condition code is always set. Refer to the individual instruction descriptions in Chapter 4 for the value of the result and of the condition codes. The type code
pushed on the stack is 6 (SRM$K_DEC_OVF_T).

6.4.1.7 Subscript Range Trap - A subscript range trap is an exception that indicates that the last instruction was an INDEX instruction with a subscript operand that failed the range check. The value of the subscript operand is lower than the low operand or greater than the high operand. The result is stored in indexout, and the condition codes are set as if the subscript were within range. The type code pushed on the stack is 7 (SRM$K_SUB_RNG_T).

6.4.1.8 Floating Overflow Fault - A floating overflow fault is an exception that indicates that the last instruction executed resulted in an exponent greater than the largest representable exponent for the data type after normalization and rounding. The destination was unaffected and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with FPD set (see Chapter 4 for details). The type code pushed on the stack is 8 (SRM$K_FLT_OVF_F).

6.4.1.9 Divide By Zero Floating Fault - A floating divide by zero fault is an exception that indicates that the last instruction executed had a floating zero divisor. The quotient operand was unaffected and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. The type code pushed on the stack is 9 (SRM$K_FLT_DIV_F).

6.4.1.10 Floating Underflow Fault - A floating underflow fault is an exception that indicates that the last instruction executed resulted in an exponent less than the smallest representable exponent for the data type after normalization and rounding and that floating underflow was enabled (FU set). The destination operand is unaffected. The saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with FPD set (see Chapter 4 for details). The type code pushed on the stack is 10 (SRM K_FLT_UND_F).
6.4.2 Memory Management Exceptions

6.4.2.1 Access Control Violation Fault - An access control violation fault is an exception indicating that the process attempted a reference not allowed at the access mode at which the process was operating. See Chapter 5, Memory Management, for a description of the information pushed on the stack as parameters. Software may restart the process after changing the address translation information.

6.4.2.2 Translation Not Valid Fault - A translation not valid fault is an exception indicating that the process attempted a reference to a page for which the valid bit in the page table was not set. See Chapter 5, Memory Management, for a description of the information pushed on the stack as parameters. Note that if a process attempts to reference a page for which the page table entry specifies both Not Valid and Access Violation, an Access Control Violation Fault occurs.
6.4.3 Exceptions Detected During Operand Reference

6.4.3.1 Reserved Addressing Mode Fault - A reserved addressing mode fault is an exception indicating that an operand specifier attempted to use an addressing mode that is not allowed in the situation in which it occurred. No parameters are pushed.

The situations in which each specifier type is reserved are:

<table>
<thead>
<tr>
<th>SPECIFIER</th>
<th>RESERVED SITUATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Literal</td>
<td>Modify, destination, address source, or within index mode.</td>
</tr>
<tr>
<td>Register</td>
<td>Address source or within index mode.</td>
</tr>
<tr>
<td>Index Mode</td>
<td>Within index mode, or with PC as index.</td>
</tr>
</tbody>
</table>

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 condition codes, which are UNPREDICTABLE.

The reserved operand exceptions are caused by:

1. A floating point number that has the sign bit set and the exponent zero except in the POLY table (FAULT)
2. A floating point number that has the sign bit set and the exponent zero in the POLY table (FAULT; see chapter 4 for restartability)
3. POLY degree too large (FAULT)
4. Decimal string too long (ABORT)
5. Invalid digit in CVTTP, CVTSP (ABORT)
6. Bit field too wide (FAULT)

7. Invalid combination of bits in PSL restored by REI (FAULT)

8. Reserved pattern operator in EDITPC (FAULT; see Chapter 4 for restartability)

9. Incorrect source string length at completion of EDITPC (ABORT)

10. Invalid combination of bits in PSW/MASK longword during RET (FAULT)

11. Invalid combination of bits in BISPSW/BICPSW (FAULT)

12. Invalid CALLx entry mask (FAULT)

13. Invalid register number in MFPR or MTPR (FAULT)

14. Invalid combinations in PCB loaded by LDPCTX (ABORT)

15. Unaligned operand in ADAWI (FAULT)

16. Invalid register contents in MTPR instructions to some registers for some implementations (FAULT):

   SISR<31:16>'SISR<0> NEQU 0
   P0BR<1:0> NEQU 0
   P0BR LSSU 2**31
   P1BR GTRU 2**31+2**30-1
   P1BR<1:0> NEQU 0
   P1BR LSSU 2**31-2**23
   P1BR GTRU 2**31+2**30-2**23-1
   P0LR<31:27>'P0LR<23:22> NEQU 0
   P1LR<30:22> NEQU 0
   ASTLVL<2:0> GTRU 4

17. Invalid operand addresses in INSQHI, INSQTI, REMQHI, or REMQTI (FAULT)
6.4.4 Exceptions Occurring As The Consequence Of An Instruction

6.4.4.1 Opcode Reserved To DIGITAL fault - An opcode reserved to DIGITAL fault occurs when the processor encounters an opcode that is not specifically defined, or that requires higher privileges than the current mode. No parameters are pushed. Opcode FFFF (hex) will always fault.

6.4.4.2 Opcode Reserved To Customers (and CSS) Fault - An opcode reserved to customers fault is an exception that occurs when an opcode reserved to the customers or DIGITAL's Computer Special Systems group is executed. The operation is identical to the opcode reserved to DIGITAL fault except that the event is caused by a different set of opcodes, and faults through a different vector. All opcodes reserved to customers (and CSS) start with FC (hex), which is the XFC instruction. If the special instruction needs to generate a unique exception, one of the reserved to CSS/Customer vectors should be used. An example might be an unrecognized second byte of the instruction.
6.4.4.3 Compatibility Mode Exception - A compatibility mode exception is an exception that occurs when the processor is in compatibility mode. A longword of information is pushed on the stack, which contains a code as follows:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reserved opcode</td>
</tr>
<tr>
<td>1</td>
<td>BPT</td>
</tr>
<tr>
<td>2</td>
<td>IOT</td>
</tr>
<tr>
<td>3</td>
<td>EMT</td>
</tr>
<tr>
<td>4</td>
<td>TRAP</td>
</tr>
<tr>
<td>5</td>
<td>illegal instruction</td>
</tr>
<tr>
<td>6</td>
<td>odd address</td>
</tr>
</tbody>
</table>

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 on tracing). At this point, the tracing program can again re-insert the BPT instruction, restore T to its original state (usually clear), and resume. Note that if both tracing and breakpointing are in progress (i.e., if PSL<T> was set at the time of the BPT), then on the trace exception both the BPT restoration and a normal trace exception should be processed by the trace handler.
6.4.5 Tracing

A trace is an exception that occurs between instructions when trace is enabled. Tracing is used for tracing programs, for performance evaluation, or debugging purposes. It is designed so that one and only one trace exception occurs before the execution of each traced instruction. The saved PC on a trace is the address of the next instruction that would normally be executed. If a trace fault and a memory management fault (or an odd address abort during a compatibility mode instruction fetch) occur simultaneously, the order in which the exceptions are taken is UNPREDICTABLE. The trace fault for an instruction takes precedence over all other exceptions.

In order to ensure that exactly one trace occurs per instruction despite other traps and faults, the PSL contains two bits, trace enable (T) and trace pending (TP). If only one bit were used then the occurrence of an interrupt at the end of an instruction would either produce zero or two traces, depending on the design. Instead of the PSL<T> bit being defined to produce a trap after any other traps or aborts at the end of an instruction, the trap effect is implemented by copying PSL<T> to a second bit (PSL<TP>) that is actually used to generate the exception. PSL<TP> generates a fault before any other processing at the start of the next instruction.

The rules of operation for trace are:

1. At the beginning of an instruction, if TP is set then a trace fault is taken after clearing TP.

2. TP is loaded with the value of T.

3. If the instruction faults or an interrupt is serviced, PSL<TP> is cleared before the PSL is pushed. The pushed PC is set to the start of the faulting or interrupted instruction. Instruction execution is resumed at Step 1.

4. If the instruction aborts or takes an arithmetic trap, PSL<TP> is not changed before the PSL is pushed.

5. If an interrupt is serviced after instruction completion and arithmetic traps but before tracing is checked for at the start of the next instruction, then PSL<TP> is not changed before the PSL is pushed.

The routine entered by a CHMx is not traced because CHMx clears T and TP in the new PSL. However, if T was set at the beginning of CHMx the saved PSL will have both T and TP set. Trace faults resume with the instruction following the REI in the routine entered by the CHMx. An instruction following an REI will fault either if T was set when the REI was executed or if TP in the saved PSL is set; in both cases TP is set after the REI. Note that a trace fault that occurs for an instruction following an REI that sets TP will be taken with the new PSL. Thus, special care must be taken if exception or interrupt routines are traced. If the T bit is set by a BISPSW instruction, trace faults begin
with the second instruction after the BISPSW.

In addition, the CALLx instructions save a clear T, although T in the PSL is unchanged. This is done so that a debugger or trace program proceeding from a BPT fault does not get a spurious trace from the RET that matches the CALL.

The detection of reserved instruction faults occurs after the trace fault. The detection of interrupts and other exceptions can occur during instruction execution. In this case, TP is cleared before the exception or interrupt is initiated. The entire PSL (including T and TP) is automatically saved on interrupt or exception initiation and is restored at the end with an REI. This makes interrupts and benign exceptions totally transparent to the executing program.
### 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, BISPSSW, and BICPSW:

<table>
<thead>
<tr>
<th>Trace exception</th>
<th>enabled at beg (T)</th>
<th>enabled at end (T)</th>
<th>TP bit at end (TP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXX</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CHMX...REI</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>interrupt or exception...REI</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CALLx</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>RET</td>
<td>N</td>
<td>N*</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>Y*</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N*</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y*</td>
<td>Y</td>
</tr>
<tr>
<td>CHMX</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>REI (if PSL&lt;TP&gt;=0 on stack)</td>
<td>N</td>
<td>N*</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>Y*</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N*</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y*</td>
<td>Y</td>
</tr>
<tr>
<td>REI (if PSL&lt;TP&gt;=1 on stack)</td>
<td>N</td>
<td>N*</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>Y*</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N*</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y*</td>
<td>Y</td>
</tr>
<tr>
<td>BISPSSW</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>BICPSW</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>interrupt or exception</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
* = depends on PSW<T> popped from stack

6.4.5.2 Using Trace - Routines using the trace facility are termed trace handlers. They should observe the following conventions and restrictions:

1. When the trace handler performs its REI back to the traced program, it should always force the T bit on in the PSL that will be restored. This defends against programs clearing T via RET, REI, or BICPSW.

2. The trace handler should never examine or alter the TP bit when continuing tracing. The hardware flows ensure that this bit is maintained correctly to continue tracing.

3. When tracing is to be ended, both T and TP should be cleared. This ensures that no further traces will occur.

4. Tracing a service routine that completes with an REI will give a trace in the restored mode after the REI. If the program being restored to was also being traced, only one trace exception is generated.

5. If a routine entered by a CALLx instruction is executed at full speed by turning off T, then trace control can be regained by setting T in the PSW in its call frame. Tracing will resume after the instruction following the RET.

6. Tracing is disabled for routines entered by a CHMx instruction or any exception. Thus, if a CHMx or exception service routine is to be traced, a breakpoint instruction must be placed at its entry point. If such a routine is recursive, breakpointing will catch each recursion only if the breakpoint is not on the CHMx or instruction with the exception.

7. If it is desired to allow multiple trace handlers, all handlers should preserve T when turning on and off trace. They also would have to simulate traced code that alters or reads T.
6.4.6 Serious System Failures

6.4.6.1 Kernel Stack Not Valid Abort - Kernel stack not valid abort is an exception that indicates that the Kernel stack was not valid while the processor was pushing information onto the Kernel stack during the initiation of an exception or interrupt. Usually this is an indication of a stack overflow or other executive software error. The attempted exception is transformed into an abort that uses the interrupt stack. No extra information is pushed on the interrupt stack in addition to PSL and PC. IPL is raised to 1F (hex). Software may abort the process without aborting the system. However, because of the lost information, the process cannot be continued. If the Kernel Stack is not valid during the normal execution of an instruction (including CMMK 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 on SCB vectors).

6.4.6.2 Interrupt Stack Not Valid Halt - An interrupt stack not valid halt is an exception that indicates that the interrupt stack was not valid or that a memory error occurred while the processor was pushing information onto the interrupt stack during the initiation of an exception or interrupt. No further interrupt requests are acknowledged on this processor. The processor leaves the PC, the PSL, and the reason for the halt in registers so that it is available to a debugger, the normal bootstrap routine, or an optional watch dog bootstrap routine. A watch dog bootstrap can cause the processor to leave the halted state.

6.4.6.3 Machine Check Exception - A machine check exception indicates that the processor detected an internal error in itself. As usual for exceptions, this exception is taken independent of IPL. IPL is raised to 1F (hex) only if vector<1:0> 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 on SCB vectors).
6.5 SERIALIZATION OF NOTIFICATION OF MULTIPLE EVENTS

The interaction between arithmetic traps, tracing, other exceptions, and multiple interrupts is complex. In order to ensure consistent and useful implementations, it is necessary to understand this interaction at a detailed level. As an example, if an instruction is started with T=1 and TP=0, it gets an arithmetic trap, and an interrupt request is recognized, the following sequence occurs:

1. The instruction finishes, storing all its results. PSL<TP> is set at the end of this instruction since PSL<T> was set at the beginning.

2. The overflow trap sequence is initiated, pushing the PC and PSL (with TP=1), loading a new PC from the vector, and creating a new PSL.

3. The interrupt sequence is initiated, pushing the PC and PSL appropriate to the overflow trap service routine, loading a new PC from the vector, and creating a new PSL.

4. If a higher priority interrupt is noticed, the first instruction of the interrupt service routine is not executed. Instead, the PC and PSL appropriate to that routine are saved as part of initiating the new interrupt. The original interrupt service routine will then be executed when the higher priority routine terminates via REI.

5. The interrupt service routine runs, and exits with REI.

6. The overflow trap service routine runs, and exits with REI, which sets PSL<TP> since the saved PSL<TP> was set.

7. The trace fault occurs, again pushing PC and PSL but this time with TP=0.

8. Trace service routine runs, and exits with REI.

9. The next instruction is executed.
This is accomplished by the following operation between instructions:

1$: {possibly take interrupts or console halt};
PSL<TP> is not modified before PSL is saved

if PSL<TP> EQU 1 then !if trace pending, take trace fault.
begin !Trace fault takes precedence
PSL<TP> <- 0; !over other exceptions.
{initiate trace fault};
end;

{possibly take interrupts or console halt};
PSL<TP> is not modified before PSL is saved

PSL<TP> <- PSL<T>; !if trace enable, set trace pending

{go start instruction execution};
!Reserved instruction faults are taken here
!FPPD is tested here, thus TP takes
!precedence over FPPD 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<FPPD> or back up all general
register side effects};
PSL<TP> <- 0;
{initiate exception or interrupt};

if {arith trap needed and no other abort
or trap} then {initiate arith trap};
end;

!note: all instructions end by flowing
!through 1$, thus the REI from a service
!routine will return to 1$
6.6 SYSTEM CONTROL BLOCK (SCB)

The System Control Block is a page containing the vectors by which exceptions and interrupts are dispatched to the appropriate service routines.

6.6.1 System Control Block Base (SCBB)

The SCBB is a privileged register containing the physical address of the System Control Block, which must be page-aligned.

3 3 2
1 0 9 +----------------------------------------+
9 8 | Physical page address of SCB |
0 +-------------------------------+ MBZ

System Control Block Base

At bootstrap time, the contents of SCBB is UNPREDICTABLE. The actual length is implementation dependent because it represents a physical address.

6.6.2 Vectors

A vector is a longword in the SCB that is examined by the processor when an exception or interrupt occurs, to determine how to service the event.

Separate vectors are defined for each interrupting device controller and each class of exceptions. Each vector is interpreted as follows by the hardware. Bits 1:0 contain a code interpreted:

0. Service this event on the kernel stack unless already running on the interrupt stack, in which case service on the interrupt stack.

1. Service this event on the interrupt stack. If this event is an exception, the IPL is raised to 1F (hex).

2. Service this event in writable control store, passing bits 15:2 to the installation-specific microcode there. If writable control store does not exist or is not loaded, the operation is UNDEFINED. On the VAX-11/780 processor, the operation in this case is a HALT.

3. Operation UNDEFINED. Reserved to DIGITAL. On the VAX-11/780 processor, the operation is a HALT.

For codes 0 and 1, bits 31:2 contain the virtual address of the service routine, which must begin on a longword boundary and will ordinarily be
in the system space. CHMx is serviced on the stack selected by the new mode. Bits <1:0> in the CHMx vectors must be zero or the operation is UNDEFINED. On the VAX-11/780 processor, these bits are ignored in the CHMx vectors.
System Control Block (exception and interrupt vectors)

<table>
<thead>
<tr>
<th>Vector (hex)</th>
<th>Name</th>
<th>Type</th>
<th>Params</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Unused</td>
<td></td>
<td></td>
<td>Reserved to DIGITAL.</td>
</tr>
<tr>
<td>04</td>
<td>Machine Check</td>
<td>Abort/Fault/Trap</td>
<td>*</td>
<td>Processor-and error-specific information is pushed on the stack, if possible. Restartability is processor specific. If vector&lt;1:0&gt; is 1, IPL is raised to 1F(hex) and the interrupt stack is used (i.e. IS &lt;- 1).</td>
</tr>
<tr>
<td>08</td>
<td>Kernel Stack Not Valid</td>
<td>Abort</td>
<td>0</td>
<td>Serviced on the interrupt stack (i.e. IS &lt;- 1). IPL is raised to 1F (hex).</td>
</tr>
<tr>
<td>0C</td>
<td>Power Fail</td>
<td>Interrupt</td>
<td>0</td>
<td>IPL is raised to 1E (hex).</td>
</tr>
<tr>
<td>10</td>
<td>Reserved/Privileged Instruction</td>
<td>Fault</td>
<td>0</td>
<td>Opcodes reserved to DIGITAL and privileged instructions.</td>
</tr>
<tr>
<td>14</td>
<td>Customer Reserved Instruction</td>
<td>Fault</td>
<td>0</td>
<td>XFC instruction.</td>
</tr>
<tr>
<td>18</td>
<td>Reserved Operand</td>
<td>Fault/Abort</td>
<td>0</td>
<td>Type depends on circumstances. See section on reserved operand exceptions.</td>
</tr>
<tr>
<td>1C</td>
<td>Reserved Addressing Mode Fault</td>
<td></td>
<td>0</td>
<td>Virtual address causing fault is pushed onto stack. See chapter 5.</td>
</tr>
<tr>
<td>20</td>
<td>Access Control Violation Fault</td>
<td>Fault</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ADDRESS</td>
<td>DESCRIPTION</td>
<td>VECTOR</td>
<td>EXCEPTION</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------</td>
<td>--------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Translation Not Valid Fault</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Trace Pending (TP) Fault</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td>Breakpoint Instruction Fault</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Compatibility Fault/Abort</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Arithmetic Trap/Fault</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38-3C</td>
<td>Unused</td>
<td></td>
<td>Reserved to DIGITAL.</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>CHMK Trap 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>CHME Trap 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>CHMS Trap 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>CHMU Trap 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>SBI SILO Compare Interrupt 0</td>
<td></td>
<td>IPL is 19 (hex). VAX-11/780 only.</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>Corrected Memory Read Data Interrupt *</td>
<td></td>
<td>IPL is 1A (hex). Also used for Read Data Substitute on VAX-11/780. Number of parameters is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>SBI Alert Interrupt 0</td>
<td></td>
<td>IPL is 1B (hex). VAX-11/780 only.</td>
<td></td>
</tr>
<tr>
<td>5C</td>
<td>SBI Fault Interrupt 0</td>
<td></td>
<td>IPL is 1C (hex).</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Interrupt</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-----------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>Memory Write Timeout</td>
<td>Interrupt *</td>
<td>VAX-11/780 only.</td>
<td></td>
</tr>
<tr>
<td>64-80</td>
<td>Unused</td>
<td>Interrupt 0</td>
<td>IPL is 1D (hex). Number of parameters is implementation dependent.</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>Software Level 1</td>
<td>Interrupt 0</td>
<td>Reserved to DIGITAL.</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>Software Level 2</td>
<td>Interrupt 0</td>
<td>Ordinarily used for AST delivery.</td>
<td></td>
</tr>
<tr>
<td>8C</td>
<td>Software Level 3</td>
<td>Interrupt 0</td>
<td>Ordinarily used for Process Scheduling.</td>
<td></td>
</tr>
<tr>
<td>90-BC</td>
<td>Software Levels 4-F</td>
<td>Interrupt 0</td>
<td>IPL is 18 (hex).</td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>Interval Timer</td>
<td>Interrupt 0</td>
<td>Reserved to DIGITAL.</td>
<td></td>
</tr>
<tr>
<td>C4-DC</td>
<td>Unused</td>
<td>Interrupt 0</td>
<td>IPL is 17 (hex). VAX-11/750 only.</td>
<td></td>
</tr>
<tr>
<td>E0-EC</td>
<td>Unused</td>
<td>Interrupt 0</td>
<td>Reserved to CSS/Customers.</td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td>Console Storage Rec.</td>
<td>Interrupt 0</td>
<td>IPL is 17 (hex). VAX-11/750 only.</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td>Console Storage Trans.</td>
<td>Interrupt 0</td>
<td>IPL is 14 (hex).</td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td>Console Terminal Rec.</td>
<td>Interrupt 0</td>
<td>IPL is 14 (hex).</td>
<td></td>
</tr>
<tr>
<td>FC</td>
<td>Console Terminal Trans.</td>
<td>Interrupt 0</td>
<td>IPL is 14 (hex).</td>
<td></td>
</tr>
<tr>
<td>100-3FC</td>
<td>Device Vectors</td>
<td>Interrupt 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the VAX-11/780 processor, only interrupt priority levels 14 to 17 (hex) are available to a NEXUS external to the CPU, and there is a limit of 16 such NEXUS. A NEXUS is a connection on the SBI, which is the internal interconnection structure. The NEXUS vectors are assigned as follows:

- 100-13C IPL 14 (hex) NEXUS 0-15
- 1A0-17C IPL 15 (hex) NEXUS 0-15
- 1B0-1BC IPL 16 (hex) NEXUS 0-15
- 1C0-1FC IPL 17 (hex) NEXUS 0-15

In the VAX-11/750 processor, UNIBUS devices interrupt the processor directly. The vector is determined by adding 200 (hex) to the vector supplied by the device. Only SCB vectors in the range 200 to 3FC (hex) are allowed. Interrupt priority levels 14 to 17 (hex) correspond to UNIBUS levels BR4 to BR7.
6.7 STACKS

At any time, the processor is either in a process context (IS=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.

6.7.1 Stack Residency

The USER, SUPER, and EXEC stacks do not need to be resident. The kernel can bring in or allocate process stack pages as Address Translation Not Valid faults occur. However, the kernel stack for the current process, and the interrupt stack (which is process-independent) must be resident and accessible. Translation Not Valid and Access Control Violation faults occurring on references to either of these stacks are regarded as serious system failures, from which recovery is not possible.

If either of these faults occurs on a reference to the kernel stack, the processor aborts the current sequence and initiates Kernel Stack Not Valid abort on hardware level 1F (hex). If either of these faults occurs on a reference to the interrupt stack, the processor halts. Note that this does not mean that every possible reference is checked, but rather that the processor will not loop on these conditions.

It is not necessary that the kernel stack for processes other than the current one be resident, but it must be resident before a process is selected to run by the software's process dispatcher. Further, any mechanism that uses Translation Not Valid or Access Control Violation faults to gather process statistics, for instance, must exercise care not to invalidate kernel stack pages.
6.7.2 Stack Alignment

Except on CALLx instructions, the hardware makes no attempt to align the stacks. For best performance on all processors, the software should align the stack on a longword boundary and allocate the stack in longword increments. The convert byte to long (CVTBL and MOVZBL), convert word to long (CVTWL and MOVZWL), convert long to byte (CVTBL), and convert long to word (CVTDLW) instructions are recommended for pushing bytes and words on the stack and popping them off in order to keep it longword aligned.

6.7.3 Stack Status Bits

The interrupt stack bit (IS) and current mode bits in the privileged Processor Status Longword (PSL) specify which of the five stack pointers is currently in use as follows:

<table>
<thead>
<tr>
<th>IS</th>
<th>MODE</th>
<th>REGISTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ISP</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>KSP</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>ESP</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>SSP</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>USP</td>
</tr>
</tbody>
</table>

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:

\[
\text{vector}<1:0> = \begin{cases} \\
00 & 01 \\
\end{cases}
\]

\[
\text{PSL}<\text{IS}> = \begin{cases} \\
0 & 1 \\
\end{cases}
\]

Values 00 (binary) and 11 (binary) of the vector<1:0> are used for other purposes. Refer to section on SCB vectors for details.

6.7.4 Accessing Stack Registers

Reference to SP (the stack pointer) in the general registers will access one of five possible architecturally defined stack pointers: the user, supervisor, executive, kernel, or interrupt stack pointer, depending on the values of the current mode and IS bits in the PSL. Some processors
might implement these five stack pointers as five internal processor registers. On these processors, software can access any of the five stack pointers not currently selected by the current mode and IS bits in the PSL via the MTPR and MFPR instructions. Results are correct even if the stack pointer specified by the current mode and IS bits in the PSL is referenced in the processor register space by an MTPR or MFPR instruction. If the process stack pointers are implemented as registers, then these instructions are the only method for accessing the stack pointers of the current process. If the process stack pointers are kept only in the PCB, MTPR and MFPR of these registers might not access the PCB. See Chapter 9 for conventions to be followed when referencing per-process registers that are also in the processor register space.

The internal processor register numbers were chosen to be the same as PSL<26:24> (see Chapter 9). The previous stack pointer is the same as PSL<23:22> unless PSL<IS> is set. If PSL<IS> is set, the previous mode cannot be determined from the PSL since interrupts always clear PSL<23:22>. At bootstrap time, the contents of all stack pointers are UNPREDICTABLE.
6.8 INITIATE EXCEPTION OR INTERRUPT

Condition Codes (if vector<1:0> code is 0 or 1):

\[
\begin{align*}
N & \leftarrow 0; \\
Z & \leftarrow 0; \\
V & \leftarrow 0; \\
C & \leftarrow 0;
\end{align*}
\]

Exceptions:

- interrupt stack not valid
- kernel stack not valid

Description:

The handling is determined by the contents of a longword vector in the system control block which is indexed by the exception or interrupt being processed. If the processor is not executing on the interrupt stack, then the current stack pointer is saved and the new stack pointer is fetched. The old PSL is pushed onto the new stack. The PC is backed up (unless this is an interrupt between instructions or a trap) and is pushed onto the new stack. The PSL is initialized to a canonical state. IPL is changed if this is an interrupt or if it is an exception with vector<1:0> code 1. Any parameters are pushed. Except for interrupts, the previous mode in the new PSL is set to the old value of the current mode. Finally, the PC is changed to point to the longword indicated by the vector<31:2>.

Notes:

1. Interrupts are disabled during this sequence.
2. If the vector<1:0> code is invalid, the behavior is UNDEFINED.
3. On an abort, the saved condition codes are UNPREDICTABLE. On a fault or interrupt, the saved condition codes are UNPREDICTABLE; they are only saved to the extent necessary to ensure correct completion of the instruction when resumed. On an abort or fault or interrupt that sets FPD, the general registers except PC, SP and FP are UNPREDICTABLE unless the instruction description specifies a setting. If FP is the destination in this case, then it is also UNPREDICTABLE. On a Kernel Stack Not Valid abort, both SP and FP are UNPREDICTABLE. In this case, UNPREDICTABLE means unspecified; upon REI the instruction behavior and results are predictable. This implies that processes stopped with FPD set cannot be resumed on processors of a different type or engineering change level.
4. If the processor gets an Access Control Violation or a Translation Not Valid condition while attempting to push information on the kernel stack, a Kernel Stack Not Valid abort is initiated and IPL is changed to IF (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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fault</td>
<td>clear</td>
</tr>
<tr>
<td>trace</td>
<td>clear</td>
</tr>
<tr>
<td>interrupt</td>
<td>clear (if FPD set)</td>
</tr>
<tr>
<td>from PSL&lt;TP&gt; (if after traps, before trace)</td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>from PSL&lt;TP&gt;</td>
</tr>
<tr>
<td>trap</td>
<td>from PSL&lt;TP&gt;</td>
</tr>
<tr>
<td>CHMx</td>
<td>from PSL&lt;TP&gt;</td>
</tr>
<tr>
<td>BPT, XFC</td>
<td>clear</td>
</tr>
<tr>
<td>reserv.instr.</td>
<td>clear</td>
</tr>
</tbody>
</table>

7. The value of PC that is saved on the stack points to the following:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fault</td>
<td>instruction faulting next instruction to execute</td>
</tr>
<tr>
<td>trace</td>
<td>i.e. instruction at the beginning of which the trace fault was taken.</td>
</tr>
<tr>
<td>interrupt</td>
<td>instruction interrupted or next instruction to execute</td>
</tr>
<tr>
<td>abort</td>
<td>instruction aborting or detecting Kernel Stack Not Valid (not ensured on machine check)</td>
</tr>
<tr>
<td>trap</td>
<td>next instruction to execute</td>
</tr>
<tr>
<td>CHMx</td>
<td>next instruction to execute</td>
</tr>
<tr>
<td>BPT, XFC</td>
<td>BPT, XFC instruction</td>
</tr>
<tr>
<td>reserv.instr.</td>
<td>reserv.instr.</td>
</tr>
</tbody>
</table>

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 LDPCCTX and SVPCCTX always fetch and store in the PCB, see Chapter 7. MEPR 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:

\[
\text{tmp1} \leftarrow (\text{SP})^+; \quad \text{! Pick up saved PC} \\
\text{tmp2} \leftarrow (\text{SP})^+; \quad \text{! and PSL} \\
\text{if } \{\text{tmp2}<\text{CUR MOD}> \text{ LSSU PSL}<\text{CUR MOD}>\} \text{ OR} \\
\{\text{tmp2}<\text{IS}> \text{ EQLU 1 AND PSL}<\text{IS}> \text{ EQLU 0}\} \text{ OR} \\
\{\text{tmp2}<\text{IS}> \text{ EQLU 1 AND tmp2}<\text{CUR MOD}> \text{ NEQU 0}\} \text{ OR} \\
\{\text{tmp2}<\text{IS}> \text{ EQLU 1 AND tmp2}<\text{IPL}> \text{ EQLU 0}\} \text{ OR} \\
\{\text{tmp2}<\text{IPL}> \text{ GTRU 0 AND tmp2}<\text{CUR MOD}> \text{ NEQU 0}\} \text{ OR} \\
\{\text{tmp2}<\text{PRV MOD}> \text{ LSSU tmp2}<\text{CUR MOD}>\} \text{ OR} \\
\{\text{tmp2}<\text{IPL}> \text{ GTRU PSL}<\text{IPL}>\} \text{ OR} \\
\{\text{tmp2}<\text{PSL MBZ}> \text{ NEQU 0}\} \text{ then } \{\text{reserved operand fault}\}; \\
\text{if } \{\text{tmp2}<\text{CM}> \text{ EQLU 1}\} \text{ AND} \\
\{\{\text{tmp2}<\text{FPD,IS,DV,PU,IV}> \text{ NEQU 0}\} \text{ OR} \\
\{\text{tmp2}<\text{CUR MOD}> \text{ NEQU 3}\}\} \text{ then } \{\text{reserved operand fault}\}; \\
\text{if PSL}<\text{IS}> \text{ EQLU 1 then ISP } \leftarrow \text{ SP} \quad \text{! Save old stack pointer} \\
\text{else PSL}<\text{CUR MOD}> \text{ SP } \leftarrow \text{ SP}; \\
\text{if PSL}<\text{TP}> \text{ EQLU 1 then tmp2}<\text{TP}> \leftarrow 1; \quad \text{! TP } \leftarrow \text{ TP or stack TP} \\
\text{PC } \leftarrow \text{ tmp1}; \\
\text{PSL } \leftarrow \text{ tmp2}; \\
\text{if PSL}<\text{IS}> \text{ EQLU 0 then} \\
\text{begin} \\
\text{SP } \leftarrow \text{ PSL}<\text{CUR MOD}> \text{ SP}; \quad \text{! Switch stack} \\
\text{if PSL}<\text{CUR MOD}> \text{ GEQU ASTLVL} \quad \text{! Check for AST delivery} \\
\text{then } \{\text{request interrupt at IPL 2}\}; \\
\text{end}; \\
\{\text{check for software interrupts}\}; \\
\{\text{clear instruction look-ahead}\}
\]

Condition Codes:

\[
\text{N } \leftarrow \text{ saved PSL}<3>; \\
\text{Z } \leftarrow \text{ saved PSL}<2>; \\
\text{V } \leftarrow \text{ saved PSL}<1>; \\
\text{C } \leftarrow \text{ saved PSL}<0>;
\]

Exceptions:

\{reserved operand\}

Opcodes:

\[02 \quad \text{REI} \quad \text{Return from Exception or Interrupt}\]
Description:

A longword is popped from the current stack and held in a temporary PC. A second longword is popped from the current stack and held in a temporary PSL. Validity of the popped PSL is checked. The current stack pointer is saved and a new stack pointer is selected according to the new PSL CUR_MOD and IS fields (see section on Stack Status Bits). The level of the highest privilege AST is checked against the current mode to see whether a pending AST can be delivered; refer to chapter 7. Execution resumes with the instruction being executed at the time of the exception or interrupt. Any instruction lookahead in the processor is reinitialized.

Notes:

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

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

3. The non-interrupt stack pointers may be fetched and stored either in privileged registers or in their allocated slots in the PCB. Only LDPCTX and SVPCTX always fetch and store in the PCB (see Chapter 7). MPFR and MTPR always fetch and store the pointers whether in registers or the PCB.
CHM  Change Mode

Purpose:  request services of more privileged software

Format:

    opcode  code.rw

Operation:

tmp1 <- {mode selected by opcode (K=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:

```
+---------------------------+     +---------------------------+     +---------------------------+
|                           |     |                           |     |                           |
|                     |     |                           |     |                           |
|    sign extended code   |     |    (SP)                   |     |    PC of next instruction |
|                     |     |                           |     |                           |
|    old PSL            |     |                           |     +---------------------------+
```

The destination mode selected by the opcode is used to obtain a location from the System Control Block. This location addresses the CHMx dispatcher for the specified mode. If the vector<1:0> 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

<table>
<thead>
<tr>
<th>INITIAL STATE</th>
<th>User IS=0 IPL=0</th>
<th>Super IS=0 IPL=0</th>
<th>Exec IS=0 IPL=0</th>
<th>Kernel IS=0 IPL=0</th>
<th>Kernel IS=0 IPL&gt;0</th>
<th>Kernel IS=1 IPL&gt;0</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>User IS=0 IPL=0</td>
<td>CHMU REI</td>
<td>CHMS</td>
<td>CHME</td>
<td>CHMK</td>
<td>Inter(0)</td>
<td>Excep(1)</td>
<td>impossible</td>
</tr>
<tr>
<td>Super IS=0 IPL=0</td>
<td>REI*</td>
<td>CHMU,S REI</td>
<td>CHME</td>
<td>CHMK</td>
<td>Inter(0)</td>
<td>Excep(1)</td>
<td>impossible</td>
</tr>
<tr>
<td>Exec IS=0 IPL=0</td>
<td>REI*</td>
<td>REI*</td>
<td>CHMU,S,E REI</td>
<td>CHMK</td>
<td>Inter(0)</td>
<td>Excep(1)</td>
<td>impossible</td>
</tr>
<tr>
<td>Kernel IS=0 IPL=0</td>
<td>REI*</td>
<td>REI*</td>
<td>REI*</td>
<td>CHMUSEK</td>
<td>MTPR IPL</td>
<td>SVPCTX</td>
<td>HALT Instr.</td>
</tr>
<tr>
<td>Kernel IS=0 IPL&gt;0</td>
<td>REI*</td>
<td>REI*</td>
<td>REI*</td>
<td>MTPR IPL</td>
<td>REI*</td>
<td>SVPCTX</td>
<td>HALT Instr.</td>
</tr>
<tr>
<td>Kernel IS=1 IPL&gt;0</td>
<td>REI*</td>
<td>REI*</td>
<td>REI*</td>
<td>LDPCTX</td>
<td>REI*</td>
<td>SVPCTX</td>
<td>HALT Instr.</td>
</tr>
</tbody>
</table>

Inter is Interrupt
Excep is Exception

(0) is vector<1:0> = 0
(1) is vector<1:0> = 1

* Any REI that increases mode can cause an interrupt request at IPL 2 for AST delivery.

Processor State Transitions
CHAPTER 7
PROCESS STRUCTURE

21-May-80 -- Rev 5

7.1 PROCESS DEFINITION

A process is a single thread of execution. It is the basic schedulable entity that is executed by the processor. A process consists of an address space and both hardware and software context. The hardware context of a process is defined by a Process Control Block (PCB) that contains images of the 14 general purpose registers, the processor status longword (PSL), the program counter (PC), the 4 per-process stack pointers, the process virtual memory defined by the base and length registers 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 context from another PCB is termed context switching. Context switching occurs as one process after another is scheduled for execution.
7.2 PROCESS CONTEXT

7.2.1 Process Control Block Base (PCBB)

The process control block for the currently executing process is pointed to by the content of the Process Control Block Base (PCBB) register, an internal privileged register. Figure 7.1 depicts the Process Control Block Base.

![Diagram of PCBB Register]

At bootstrap time, the contents of PCBB is UNPREDICTABLE.

7.2.2 Process Control Block (PCB)

The process control block (PCB) contains all of the switchable process context, collected into a compact form for ease of movement to and from the privileged internal registers. Although in any normal operating system there is additional software context for each process, the following description is limited to that portion of the PCB known to the hardware. Figure 7-2 depicts the PCB, whose contents are described in Table 7-1.
Figure 7-2 Process Control Block (PCB)
# Table 7-1

Description of Process Control Block

<table>
<thead>
<tr>
<th>Longword</th>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;31:0&gt;</td>
<td>KSP</td>
<td>Kernel Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 0 and IS = 0.</td>
</tr>
<tr>
<td>1</td>
<td>&lt;31:0&gt;</td>
<td>ESP</td>
<td>Executive Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 1.</td>
</tr>
<tr>
<td>2</td>
<td>&lt;31:0&gt;</td>
<td>SSP</td>
<td>Supervisor Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 2.</td>
</tr>
<tr>
<td>3</td>
<td>&lt;31:0&gt;</td>
<td>USP</td>
<td>User Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 3.</td>
</tr>
<tr>
<td>4-17</td>
<td>&lt;31:0&gt;</td>
<td>R0-R11, AP, FP</td>
<td>General registers R0 through R11, AP, FP.</td>
</tr>
<tr>
<td>18</td>
<td>&lt;31:0&gt;</td>
<td>PC</td>
<td>Program Counter.</td>
</tr>
<tr>
<td>19</td>
<td>&lt;31:0&gt;</td>
<td>PSL</td>
<td>Program Status Longword.</td>
</tr>
<tr>
<td>20</td>
<td>&lt;31:0&gt;</td>
<td>P0BR</td>
<td>Base register for page table describing process virtual addresses from 0 to 2**30-1. See chapter 5.</td>
</tr>
<tr>
<td>21</td>
<td>&lt;21:0&gt;</td>
<td>P0LR</td>
<td>Length register for page table located by P0BR. Describes effective length of page table. See chapter 5.</td>
</tr>
<tr>
<td>21</td>
<td>&lt;23:22&gt;</td>
<td>MBZ</td>
<td>Must be zero.</td>
</tr>
</tbody>
</table>
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>  PLBR  Base register for page table describing process virtual addresses from $2^{30}$ to $2^{31}-1$. See chapter 5.

23  <21:0>  PLLR  Length register for page table located by PLBR. 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 to permit their behavior to be observed without interference from other system activity.
Software symbols for these locations consist of the prefix PTX$SL and the mnemonic. For example, the PCB offset to R3 is PTX$SL_R3. Exceptions are longwords 21 and 23, for which the software symbols are:

\[
\begin{align*}
\text{PTX$SL\_P0LRASTL} & \quad \text{longword 21} \\
\text{PTX$SL\_PILRPME} & \quad \text{longword 23}
\end{align*}
\]

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. Figure 7.3 depicts the AST Level Register.

```
 3
 1
 +------------------------+ 3 2 0
 | ignored; returns 0 | +----+
 | | LVL |
 +------------------------+

(read/write)
Figure 7-3 AST Level Register
```

An MTIP src,#ASTLVL with src<2:0> GEQU 5 results in a reserved operand fault. At bootstrap time, the contents of ASTLVL is 4. Figure 7.4 depicts the Performance Monitor Enable (PME) Register.

```
 3
 1
 +---------------------+ 1 0
 |                    | +----+
 | MBZ                | | P |
 |                   | | M |
 +---------------------++ E

(read/write)
Figure 7-4 Performance Monitor Enable Register
```

At bootstrap time, PME is cleared.
7.3 ASYNCHRONOUS SYSTEM TRAPS (AST)

Asynchronous system traps are a technique for notifying a process of events that are not synchronized with its execution and initiating processing for asynchronous events with the least possible delay. This delay in delivery of the AST may be due to either process non-residence or an access mode mismatch. The efficient handling of AST's in VAX-11 requires some hardware assistance to detect changes in access mode (current access mode in PSL). A process in any of the four execution access modes (kernel, exec, super, and user) may receive AST's; however, an AST for a less privileged access mode must not be permitted to interrupt execution in a more protected access mode. Since outward access mode transitions occur only in the REI instruction, comparison of the current access mode field is made with a privileged register (ASTLVL) containing the most privileged access mode number for which an AST is pending. If the new access mode is greater than or equal to the pending ASTLVL, an IPL 2 interrupt is triggered to cause delivery of the pending AST.

General Software Flow for AST processing:

1. An event associated with an AST causes software enqueuing of an AST control block to the software PCB and the software sets the ASTLVL field in the hardware PCB to the most privileged access mode for which an AST is pending. If the target process is currently executing, the ASTLVL privileged register also has to be set.

2. When an REI instruction detects a transition to an access mode that can be interrupted by a pending AST, an IPL 2 interrupt is triggered to cause delivery of the AST. Note that the REI instruction does not make pending AST checks while returning to a routine executing on the interrupt stack.

3. The (IPL 2) interrupt service routine should compute the correct new value for ASTLVL that prevents additional AST delivery interrupts while in kernel mode and move that value to the PCB and the ASTLVL register before lowering IPL and actually dispatching the AST. This interrupt service routine normally executes on the kernel stack in the context of the process receiving the AST.

4. At the conclusion of processing for an AST, the ASTLVL is recomputed and moved to the PCB and ASTLVL register by software.
7.4 PROCESS STRUCTURE INTERRUPTS

Two of the software interrupt priorities are reserved for process structure software.

They are:

(IPL 2) - AST delivery interrupt.

This interrupt is triggered by a REI that detects PSL<CUR_MOD> GEQV ASTLVL and indicates that a pending AST may now be delivered for the currently executing process.

(IPL 3) - Process scheduling interrupt.

This interrupt is only triggered by software to allow the software running at IPL 3 to cause the currently executing process to be blocked and the highest priority executable process to be scheduled.

7.5 PROCESS STRUCTURE INSTRUCTIONS

Process scheduling software must execute on the interrupt stack (PSL<IS> set) in order to have a non-context-switched stack available for use. If the scheduler were running on a process's kernel stack, then any state information it had there would disappear when a new process is selected. Running on the interrupt stack can occur as the result of the interrupt origin of scheduling events, however some synchronous scheduling requests such as a WAIT service may want to cause rescheduling without any interrupt occurrence. For this reason, the Save Process Context (SVPCTX) instruction can be executed while on either the kernel or the interrupt stack and forces a transition to execution on the interrupt stack.

All of the process structure instructions are privileged and require kernel mode.
LDPCTX  Load Process Context

Purpose:  restore register and memory management context

Format:

opcode

Operation:

if PSL < CUR_MOD > NEQU 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);
tmpl <- (PCB+80);
if {tmpl<31:30> NEQU 2} OR {tmpl<1:0> NEQU 0} then
  {reserved operand abort};
P0BR <- tmpl;
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>;
tmpl <- (PCB+88);
tmp2 <- tmpl + 2**23;
if {tmp2<31:30> NEQU 2} OR {tmp2<1:0> NEQU 0} then
  {reserved operand abort};
P1BR <- tmpl;
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
begin
ISP <- SP;
{interrupts off};
PSL<IS> <- 0;
SP <- (PCB);
{interrupts on};
end;
-(SP) <- (PCB+76);
get KSP
-(SP) <- (PCB+72);
push PSL
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, LDPCTX loads the internal registers from the PCB. Processors that do not keep a copy of all four per-process stack pointers in internal registers, keep only the current access mode register in an internal register and switch this with the PCB contents whenever the current access mode field changes.

2. Some implementations may not perform some or all of the reserved operand checks.
SVPCtx  Save Process Context

Purpose:  save register context

Format:

opcode

Operation:

if PSL<CUR_MOD> NEQU 0 then
  {privileged instruction fault};
  !PCB is located by physical address in PCBB
  if {internal registers for stack pointers} then
    begin
      (PCB) <- KSP;
      (PCB+4) <- ESP;
      (PCB+8) <- SSP;
      (PCB+12) <- USP;
      end;
    (PCB+16) <- R0;
    (PCB+20) <- R1;
    (PCB+24) <- R2;
    (PCB+28) <- R3;
    (PCB+32) <- R4;
    (PCB+36) <- R5;
    (PCB+40) <- R6;
    (PCB+44) <- R7;
    (PCB+48) <- R8;
    (PCB+52) <- R9;
    (PCB+56) <- R10;
    (PCB+60) <- R11;
    (PCB+64) <- AP;
    (PCB+68) <- FP;
    (PCB+72) <- (SP)+;  !pop PC
    (PCB+76) <- (SP)+;  !pop PSL
  if PSL<IS> EQULU 0 then
    begin
      PSL<IPL> <- MAXU(1, PSL<IPL>);
      (PCB) <- SP;  !save KSP
      KSP <- SP;
      {interrupts off};
      PSL<IS> <- 1;
      SP <- ISP;
      {interrupts on};
      end;

Condition Codes:

N <- N;
Z <- Z;
V <- V;
C <- C;
Exceptions:

privileged instruction

Opcodes:

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

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

The following example illustrates how the process structure instructions can be used to implement process dispatching software. It is assumed that this simple dispatch routine is always entered via an interrupt.

; ENTERED VIA INTERRUPT
; IPL=3

RESCHED: SVPCTX ; Save context in PCB
  .
  .
  .<set state to runnable>
  .<and place current PCB>
  .<on proper RUN queue>
  .
  .<Remove head of highest>
  .<priority, non-empty,>
  .<RUN queue.>
MTPR @#PHYS PCB, PCBB ; Set physical PCB address
  ; in PCB
LDPCTX ; Load context from PCB
  ; For new process
REI ; Place process in execution
CHAPTER 8
SYSTEM ARCHITECTURAL IMPLICATIONS

17-June-80 -- Rev 5

8.1 INTRODUCTION

Certain portions of the VAX-11 architecture have implications on the system structure of implementations. There are four broad categories of interaction: data sharing and synchronization, restartability, interrupts and errors. Of these, data sharing is most visible to the programmer.

8.2 DATA SHARING AND SYNCHRONIZATION

The memory system must be implemented such that the granularity of access for independent modification is the byte. Note that this does not imply a maximum reference size of one byte but only that independent modifying accesses to adjacent bytes produce the same results regardless of the order of execution. For example, suppose locations 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. Seven instructions (BBSSI, BBCCI, ADAWI, INSQHI, INSQTI, REMQHI, REMQTI) are provided to allow the programmer to control ("interlock") access to a control variable. These interlocked instructions must be implemented in such a way that read, test, modify, and write happen while other processors and I/O devices are locked out of performing interlocked operations on the same control variable. This is termed an interlocked sequence. Only interlocking operations are locked out by the interlock. On the VAX-11/780, the SBI primitive operations are interlock read and interlock write. The interlocked read operation sets the interlock, and the interlocked write releases it.
BBSSI and BBCCI instructions use hardware provided primitive operations to make a read reference, then test, and then make a write reference to a single bit within a single byte in an interlocked sequence. The ADAOI 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 ADAOI instruction takes the hardware lock on the read of the .mw operand (the second operand which is the one being modified).

The INSQ and REMQ instructions provide a series of longword reads and writes in an uninterruptible sequence to allow queues to be maintained without other interlocks in a uniprocessor system.

The INSQHI, INSQTI, REMQHI, and REMQTI instructions use an interlock on the queue header to allow queues to be maintained consistently in a multiprocessor system.

In order to provide a functionality upon which some UNIBUS peripheral devices rely, processors must insure that all instructions making byte or word sized modifying references (.mb and .mw) use the DATIP - DATOB functions when the operand physical address selects a UNIBUS device. This constraint does not apply to longword, quadword, field, all floating, or string operations if implemented using byte or word modifying references. This constraint also does not apply to instructions precluded from I/O space references (see Appendix A).

In a multiprocessor system, any software clearing PTE<Y> or changing the protection code of a page table entry for system space such that it issues a MTPR xxx,#TBIS must arrange for all other processors to issue a similar TBIS. The original processor must wait until all the other processors have completed their TBIS before it allows access to the system page.

8.3 CACHE

A hardware implementation may include a mechanism to reduce access time by making local copies of recently used memory contents. Such a mechanism is termed a cache. A cache must be implemented in such a way that its existence is transparent to software (except for timing and error reporting/control/recovery). In particular, the following must be true:

1. Program writes to memory followed by starting a peripheral output transfer must output the updated value.
2. Completing a peripheral input transfer followed by the program reading of memory must read the input value.
3. A write or modify followed by a HALT on one processor followed by a read or modify on another processor must read the updated value.
4. A write or modify followed by a power failure followed by restoration of power followed by a read or modify must read the updated value provided that the duration of the power failure does not exceed the maximum non-volatile period of the main memory.

5. In multiprocessor systems, access to variables shared between processors must be interlocked by software executing one of the interlocked instructions (BBSSI, BBCCI, ADAWI, INSQHI, INSQTI, REMQHI, REMQTI).

6. Valid accesses to I/O registers must not be cached.

On the VAX-11/780, this is achieved by a cache that writes through to memory and that watches the memory bus for all external writes to memory.

At bootstrap time, the cache must be either empty or valid.

8.4 RESTARTABILITY

The VAX-11 architecture requires that all instructions be restartable after a fault or interrupt that terminated execution before the instruction was completed. Generally, this means that modified registers are restored to the value they had at the start of execution. For some complex or iterative instructions, indicated in Chapter 4, intermediate results are stored in the general registers. In the latter case memory contents may have been altered but the former case requires that no operand be written unless the instruction can be completed. For most instructions with only a single modified or written operand, this implies special processing only when a multibyte operand spans a protection boundary making it necessary to test accessibility of both parts of the operand.

In order that instructions which store intermediate results in the general registers not compromise system integrity, they must insure that any addresses stored or used are virtual addresses, subject to protection checking, and that any state information stored or used cannot result in a non-interruptable or non-terminating sequence.

Instruction operands that are peripheral device registers having access side effects may produce UNPREDICTABLE results due to instruction restarting after faults or interrupts. In order that software may dependably access peripheral device registers, instructions used to access them must not permit a fault or interrupt after the first I/O space access. The instructions and addressing modes that can be used to meet this condition are listed in Appendix A, "INSTRUCTIONS USABLE TO REFERENCE I/O SPACE."

Memory modifications produced as a side effect of instruction execution, e.g., memory access statistics, are specifically excluded from the constraint that memory not be altered until the instruction can be
completed.

Instructions that abort are constrained only to insure memory protection (e.g., registers can be changed).

8.5 INTERRUPTS

Underlying the VAX-11 architectural concept of an interrupt is the notion that an interrupt request is a static condition, not a transient event, which can be sampled by a processor at appropriate times. Further, if the need for an interrupt disappears before a processor has honored an interrupt request, the interrupt request can be removed (subject to implementation dependent timing constraints) without consequence.

In order that software be able to operate deterministically it is necessary that any instruction changing the processor priority (IPL) such that a pending interrupt is enabled must allow the interrupt to occur before executing the next instruction that would have been executed had the interrupt not been pending.

Similarly, instructions that generate requests at the software interrupt levels (See Chapter 5) must allow the interrupt to occur, if processor priority permits, before executing the apparently subsequent instruction.

8.6 ERRORS

Processor errors, if not inconsistent with instruction completion, should create high priority interrupt requests. Otherwise, they must terminate instruction execution with an exception (fault, trap or abort), in which case there may also be an associated interrupt request.

Error notification interrupts may be delayed from the apparent completion of the instruction in execution at the time of the error but if enabled, the interrupt must be requested before processor context is switched, priority permitting.

An example of a case where both an interrupt and an exception are associated with the same event occurs when the VAX-11/780 instruction buffer gets a read data substitution (i.e. read memory data error). In this case the interrupt request associated with error will not be taken if the priority of the running program is high, but an abort will occur when an attempt is made to execute the instruction. However, the interrupt is still pending and will be taken when the priority is lowered.
8.7 I/O STRUCTURE

8.7.1 Introduction

The VAX-11 I/O architecture is very similar to the PDP-11 structure, the principal difference being the method by which processor registers (such as the PSL) are accessed (see Chapter 9). Peripheral device control/status and data registers appear at locations in the physical address space, and can therefore be manipulated by normal memory reference instructions. On the VAX-11/780 implementation, this I/O space occupies the upper half of the physical address space and is $2^{29}$ bytes in length. Use of general instructions permits all the virtual address mapping and protection mechanisms described in Chapter 5 to be used when referencing I/O registers. Note: Implementations that include a cache feature must suppress caching for references in the I/O space.

For any member of the VAX-11 series implementing the UNIBUS, there will be one or more areas of the I/O physical address space each $2^{18}$ bytes in length, which "maps through" to the UNIBUS addresses. The collection of these areas is referred to as the UNIBUS space.

8.7.2 Constraints On I/O Registers

The following is a list of both hardware and programming constraints on I/O registers. These items affect both hardware register design and programming considerations.

1. The physical address of an I/O register must be an integral multiple of the register size in bytes, (which must be a power of two); i.e., all registers must be aligned on natural boundaries.

2. References using a length attribute other than the length of the register and/or unaligned references may produce UNPREDICTABLE results. For example a byte reference to a word-length register will not necessarily respond by supplying or modifying the byte addressed.

3. In all peripheral devices, error and status bits that may be asynchronously set by the device must be cleared by software writing a "1" to that bit position and not affected by writing a "0". This is to prevent clearing bits that may be asynchronously set between reading and writing a register.

4. Only byte and word references of a read-modify-write (i.e., ".mb" or ".mw") type in UNIBUS I/O spaces are guaranteed to interlock correctly. References in the I/O space other than in UNIBUS spaces are UNDEFINED with respect to interlocking. This includes the BBSI and BBCCI instructions.
5. String, quad, octa, F-floating, D-floating, G-floating, H-floating, and field references in the I/O space result in UNDEFINED behavior.
CHAPTER 9
PRIVILEGED REGISTERS

13-May-81 -- Rev 5.2

9.1 PROCESSOR REGISTER SPACE

The processor register space (PRS) provides access to many types of CPU control and status registers such as the memory management base registers, the PSL, and the multiple stack pointers. These registers are explicitly accessible only by the Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions which require kernel mode privileges.

All the internal processor registers are summarized in the tables at the end of this section. Those which need further explanation are described below. Reference to general registers means 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 PER-PROCESS Registers AND CONTEXT SWITCHING

There are several per-process registers which are loaded from the PCB during a context load operation and, with the exception of the memory mapping registers and AST level, written back to the PCB during a context save operation (see Chapter 7). Some implementations may copy some or all of these registers from the PCB into scratchpad registers and write them back into the PCB during a context save operation. Other implementations may retain the registers in main memory in the PCB.

For this reason, reading or writing any of these registers via the MFPR or MTPR instruction, or through reference to SP, may or may not read or write the register copy in the current PCB, depending on the implementation. Likewise modifying one of these registers in the PCB will not necessarily update the register which appears in the register space or SP.
An implementation may retain some or all per-process internal registers only in the PCB. In this case, MTPR and MFPR for these registers must access the corresponding PCB location. However, implementations that have internal registers in hardware scratchpads are not required to access the corresponding PCB locations for MTPR and MFPR.

9.3 STACK POINTER IMAGES

Reference to SP (the stack pointer) in the general registers will access one of five possible stack pointers; the user, supervisor, executive, kernel, or interrupt stack pointer, depending on the values of the current mode and IS bits in the PSL (see Chapter 5). Additionally, software can access any of the five stack pointers (including the one currently selected by the current mode and IS bits in the PSL) via the MTPR and MFPR instructions (even on processors that implement the KSP, SSP, ESP, or USP only in the PCB) Results are correct even if the stack pointer specified by the current mode and IS bits in the PSL is referenced in the PRS by an MTPR or MFPR instruction. This means that a MFPR/MTPR to the KSP (if IS=0) or the ISP (if IS=1) is equivalent to a MOVL from/to the SP.
9.4  MTPR AND MFPR INSTRUCTIONS

MTPR  Move To Processor Register

Format:

opcode  src.rl, procreg.rl

Operation:

if PSL <CUR_MOD> NEQ 0 then {reserved
  instruction fault};
PRS[procreg] <- src;

Condition Codes:

N <- src LSS 0;  % if register is replaced
Z <- src EQL 0;
V <- 0;  % except TBCHK register (see Chapter 5)
C <- C;

N <- N;  % if register is not replaced
Z <- Z;
V <- V;
C <- C;

Exceptions:

reserved operand fault
reserved instruction fault

Opcode:

DA  MTPR  Move To Processor Register

Description:

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

Notes:

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

2. A reserved instruction fault occurs if instruction execution is
   attempted in other than kernel mode.
3. A reserved operand fault occurs on a move to a read only register.
MPFR       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
    Z <- Z;
    V <- V;
    C <- C;

Exceptions:

    reserved operand fault
    reserved instruction fault

Opcode:

    DB      MPFR       Move From Processor Register

Description:

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

Notes:

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

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

3. A reserved operand fault occurs on a move from a write only
   register.
## 9.5 VAX-11 SERIES REGISTERS

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Mnemonic</th>
<th>Number</th>
<th>Type</th>
<th>Scope</th>
<th>Init?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Stack Pointer</td>
<td>KSP</td>
<td>0</td>
<td>R/W</td>
<td>PROC</td>
<td>--</td>
</tr>
<tr>
<td>Executive Stack Pointer</td>
<td>ESP</td>
<td>1</td>
<td>R/W</td>
<td>PROC</td>
<td>--</td>
</tr>
<tr>
<td>Supervisor Stack Pointer</td>
<td>SSP</td>
<td>2</td>
<td>R/W</td>
<td>PROC</td>
<td>--</td>
</tr>
<tr>
<td>User Stack Pointer</td>
<td>USP</td>
<td>3</td>
<td>R/W</td>
<td>PROC</td>
<td>--</td>
</tr>
<tr>
<td>Interrupt Stack Pointer</td>
<td>ISP</td>
<td>4</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>P0 Base Register</td>
<td>P0BR</td>
<td>8</td>
<td>R/W</td>
<td>PROC</td>
<td>--</td>
</tr>
<tr>
<td>P0 Length Register</td>
<td>P0LR</td>
<td>9</td>
<td>R/W</td>
<td>PROC</td>
<td>--</td>
</tr>
<tr>
<td>P1 Base Register</td>
<td>P1BR</td>
<td>10</td>
<td>R/W</td>
<td>PROC</td>
<td>--</td>
</tr>
<tr>
<td>P1 Length Register</td>
<td>P1LR</td>
<td>11</td>
<td>R/W</td>
<td>PROC</td>
<td>--</td>
</tr>
<tr>
<td>System Base Register</td>
<td>SBR</td>
<td>12</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>System Limit Register</td>
<td>SLR</td>
<td>13</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Process Control Block Base</td>
<td>PCBB</td>
<td>16</td>
<td>R/W</td>
<td>PROC</td>
<td>--</td>
</tr>
<tr>
<td>System Control Block Base</td>
<td>SCBB</td>
<td>17</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Interrupt Priority Level</td>
<td>IPL</td>
<td>18</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>AST Level</td>
<td>ASTLVL</td>
<td>19</td>
<td>R/W</td>
<td>PROC</td>
<td>yes</td>
</tr>
<tr>
<td>Software Interrupt Request</td>
<td>SIRR</td>
<td>20</td>
<td>W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Software Interrupt Summary</td>
<td>SISR</td>
<td>21</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Interval Clock Control</td>
<td>ICSS</td>
<td>24</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Next Interval Count</td>
<td>NICR</td>
<td>25</td>
<td>W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Interval Count</td>
<td>ICR</td>
<td>26</td>
<td>R</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Time of Year (optional)</td>
<td>TODR</td>
<td>27</td>
<td>R/W</td>
<td>CPU</td>
<td>no</td>
</tr>
<tr>
<td>Console Receiver C/S</td>
<td>RXCS</td>
<td>32</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Console Receiver D/B</td>
<td>RXDB</td>
<td>33</td>
<td>R</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Console Transmit C/S</td>
<td>TXCS</td>
<td>34</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Console Transmit D/B</td>
<td>TXDB</td>
<td>35</td>
<td>W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Memory Management Enable</td>
<td>MAPEN</td>
<td>56</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Trans. Buf. Invalidate All</td>
<td>TBIA</td>
<td>57</td>
<td>W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Trans. Buf. Invalidate Single</td>
<td>TBIS</td>
<td>58</td>
<td>W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Performance Monitor Enable</td>
<td>PMR</td>
<td>61</td>
<td>R/W</td>
<td>PROC</td>
<td>yes</td>
</tr>
<tr>
<td>System Identification</td>
<td>SID</td>
<td>62</td>
<td>R</td>
<td>CPU</td>
<td>no</td>
</tr>
<tr>
<td>Translation Buffer Check</td>
<td>TBCHK</td>
<td>63</td>
<td>W</td>
<td>CPU</td>
<td>--</td>
</tr>
</tbody>
</table>
9.5.1 System Identification Register (SID)

The SID is a read only constant register that specifies the processor type. The entire SID register is included in the error log and the type field may be used by software to distinguish processor types.

<table>
<thead>
<tr>
<th>3 2 2</th>
<th>1 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>type specific</td>
<td>(read only)</td>
</tr>
</tbody>
</table>

System Identification Register

Type
A unique number assigned by engineering to identify a specific processor:

0 = Reserved to DIGITAL (error)
1 = VAX-11/780
2 = VAX-11/750
3 = VAX-11/730
4 through 127 = Reserved to DIGITAL
128 through 255 = Reserved to CSS and customers

For the VAX-11/780, the type specific format is:

<table>
<thead>
<tr>
<th>2 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 5 4 2 1</td>
</tr>
</tbody>
</table>

For the VAX-11/750, the type specific format is:

<table>
<thead>
<tr>
<th>2 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 6 5 8 7</td>
</tr>
</tbody>
</table>

| microcode rev | hardware rev |
9.5.2 Console Terminal Registers

The console terminal is accessed through four internal registers. Two are associated with receiving from the terminal and two with writing to the terminal. In each direction there is a control/status register and a data buffer register.

Console Receive Control/Status (RXCS)

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 MPFR #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)

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 executed, 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.

9.5.2.1 VAX-11/780 console register implementation - RXDB
Select Field Values (in Hex)

<table>
<thead>
<tr>
<th>Select Code</th>
<th>Device</th>
<th>Data Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operator's Terminal</td>
<td>0 thru 7F - ASCII Data</td>
</tr>
<tr>
<td>1</td>
<td>Drive 0 (Data)</td>
<td>0 thru FF - Binary Data</td>
</tr>
<tr>
<td>2</td>
<td>Function Complete</td>
<td>(Status)</td>
</tr>
<tr>
<td>9</td>
<td>Drive 0 (Command)</td>
<td>0 = Read Sector</td>
</tr>
<tr>
<td>F</td>
<td>Misc. Communication</td>
<td>1 = Write Sector</td>
</tr>
</tbody>
</table>
<pre><code>                            | 2 = Read Status                     |
                            | 3 = Write Deleted Data Sector       |
                            | 4 = Cancel Function                 |
                            | 5 = Protocol Error                  |
                            | 1 = Software Done                   |
                            | 2 = Boot CPU                        |
                            | 3 = Clear Warm-start flag           |
                            | 4 = Clear Cold-start flag           |
</code></pre>

Code 5 (Protocol Error), is sent by the console when one of the following occurs:

1. Another load device command (except for Cancel Function) is issued by the OS before a previous command is completed.

2. The console gets a 'Drive 0 (DATA)' when expecting a command.
9.5.2.1.1 Status Byte Definition - The Status Byte is used by VMS to determine the success or failure of a Read or Write operation. The Status Byte is sent to the OS at the completion of a Read, Write, or Read Status operation. The Select code is always 'Function Complete' (code 2). The Status Bit assignments are as follows:

RXDB

<table>
<thead>
<tr>
<th>3</th>
<th>2 2</th>
<th>1 1</th>
<th>1 1</th>
<th>1 1</th>
<th>2 1</th>
<th>8</th>
<th>7 6</th>
<th>2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBZ</td>
<td>MBZ</td>
<td>MBZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CODE '2'</td>
<td></td>
<td></td>
<td>CRC ERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>PARITY ERROR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INI DONE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DELETED DATA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ERROR</td>
<td></td>
</tr>
</tbody>
</table>

The Status Bit assignments are identical to those supplied by the Floppy controller, excepting Bit 7. Bit 7 corresponds to Bit 15 of the Floppy's 'RXCS' Register.

9.5.3 Clock Registers

The clocks consist of a time of year clock and an interval clock. The time of year clock is used to measure the duration of power failures and is required for unattended restart after a power failure. The interval clock is used for accounting, for time dependent events, and to maintain the software date and time.

9.5.3.1 Time-of-Year Clock -

The time-of-year clock consists of one longword register. The register forms an unsigned 32-bit binary counter that is driven by a precision clock source with at least 0.0025% accuracy (approximately 65 seconds per month). The least significant bit of the counter represents a resolution of 10 milliseconds. Thus, the counter cycles to 0 after approximately 497 days.
The counter has an optional battery back-up power supply sufficient for at least 100 hours of operation, and the clock does not gain or lose any ticks during transition to or from stand-by power. The battery is recharged automatically. If the battery has failed, so that time is not accurate, then the register is cleared upon power up. One of two things then happens:

1. The register starts counting from 0. Thus, if software initializes this clock to a value corresponding to a large time (e.g., a month), it can check for loss of time after a power restore by checking the clock value. This is the VAX-11/780 implementation.

2. The register stays at 0 until the software writes a non-zero value into it. It counts only when it contains a non-zero value. This is the VAX-11/750 implementation.

```
+---------------------------------- 0
| time of year since setting |
+-----------------------------
(readonly/write)
```

Time of Year (TODR)
9.5.3.2 Interval Clock - The interval clock provides an interrupt at IPL 24 at programmed intervals. The counter is incremented at 1 microsecond intervals, with at least .01% accuracy (3.64 seconds per day). The clock interface consists of three registers in the privileged register space:

```
3 1
+------------------------
| interval count         |
+------------------------+
   (read only)
interval count register (ICR)

3 1
+------------------------
| next interval count    |
+------------------------+
   (write only)
next interval (NICR)
```

```
3 3
1 0
++
|E| |I|I|S|X| |R|
|R| |N|E|G|F| MBZ |U|
|R| |T| |L|R| |N|
++
W  C
W R W W R
C W O O W
```

Interval Clock Control/Status (ICCS)

1. Interval Count - The interval register is a read only register incremented once every microsecond. It is automatically loaded from NICR upon a carry out from bit 31 (overflow) which also interrupts at IPL 24 if the interrupt is enabled.

2. Next Interval Count - The reload register is a write only register that holds the value to be loaded into ICR when it overflows. The value is retained when ICR is loaded. NICR is capable of being loaded regardless of the current values of ICR and ICCS.

3. Interval Clock Control Status (ICCS) - The ICCS register contains control and status information for the interval clock.

**RUN <0>** When set, ICR increments each microsecond. When clear, ICR does not increment automatically. At bootstrap time, run is cleared.
Privileged Registers

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 #XCI,#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 SPECIFIC REGISTERS

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Mnemonic</th>
<th>Number</th>
<th>Type</th>
<th>Scope</th>
<th>Init?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerator Control/Status</td>
<td>ACCS</td>
<td>40</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Accelerator Maintenance</td>
<td>ACCR</td>
<td>41</td>
<td>R/W</td>
<td>CPU</td>
<td>no</td>
</tr>
<tr>
<td>WCS Address</td>
<td>WCSA</td>
<td>44</td>
<td>R/W</td>
<td>CPU</td>
<td>no</td>
</tr>
<tr>
<td>WCS Data</td>
<td>WCSD</td>
<td>45</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Fault/Status</td>
<td>SBIFS</td>
<td>48</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Silo</td>
<td>SBIS</td>
<td>49</td>
<td>R</td>
<td>CPU</td>
<td>no</td>
</tr>
<tr>
<td>SBI Silo Comparator</td>
<td>SBISC</td>
<td>50</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Maintenance</td>
<td>SBIMT</td>
<td>51</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Error Register</td>
<td>SBIER</td>
<td>52</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Timeout Address</td>
<td>SBITA</td>
<td>53</td>
<td>R</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>SBI Quadword Clear</td>
<td>SBIQC</td>
<td>54</td>
<td>W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Micro Program Breakpoint</td>
<td>MBRK</td>
<td>50</td>
<td>R/W</td>
<td>CPU</td>
<td>no</td>
</tr>
</tbody>
</table>

9.6.1 VAX-11/780 Accelerator

The VAX-11/780 processor has an optional accelerator for a subset of the instructions. Two internal registers control the accelerator, ACCS and ACCR.

ACCS is the accelerator control and status register. It indicates whether an accelerator exists, controls whether it is enabled, identifies its type and reports errors and status. At bootstrap time, the type and enable are set; the errors are cleared.

```
 3 3 2 2 2 2
1 0 9 8 7 6
 1 1 1 6 5 4
     8 7 0
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

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.
Privileged Registers

VAX-11/780 SPECIFIC REGISTERS

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 microprogram counter. At bootstrap time its contents are UNPREDICTABLE.

```
  3 3  2 2  1 1 1  1 1 1
  0  4 3  6 5 4 3  9 8  0
```

<table>
<thead>
<tr>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Accelerator Maintenance Register (ACCR)

PC <0:8> NEXT MICRO PC on read. This contains the next microaddress to be executed.

MATCH MICRO PC on write. If EML is also set, then this updates the micro PC match register.

MPM <14> MICRO PC MATCH. A read only bit that is set whenever the accelerator's micro PC matches the micro PC match register. This is useful primarily as a scope sync signal.

EML <15> ENABLE MICRO PC MATCH LOAD. A write only bit that when set causes <8:0> to be loaded into the accelerator's micro PC match register.

TRAP <16:23> TRAP ADDRESS. A read/write field used by the main processor to force the accelerator to a specified micro location.
ETL <31> ENABLE TRAP ADDRESS LOAD. A write only bit that when set causes <23:16> to be loaded into the accelerator's trap address register. Subsequently, the main processor's micro code can force the accelerator to trap to this location by asserting an internal signal.

9.6.2 VAX-11/780 Micro Control Store

The VAX-11/780 processor has three registers for control/status of its microcode. Two are used for writing into any writable control store (WCS) and one is used to control micro breakpoints.

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>1 1 1 1</th>
<th>6 5 4 3 2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MBZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P</td>
<td>I</td>
<td>CTR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R RW</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Writable Control Store Address (WCSA)

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(on Write)

<table>
<thead>
<tr>
<th>3</th>
<th>8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRESENT</td>
</tr>
</tbody>
</table>

(on read)

Writable Control Store Data (WCSD)

Reading WCSD indicates which control store addresses are writable. If WCSD<4> 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.

9.6.3 SBI FAULT/STATUS REGISTER (SBIFS)

Fault Silo Lock
(set if Silo Locked due to Fault Signal)
Fault Signal
Fault Interrupt Enable
Fault Latch
Nested Error
Transmitter during Fault cycle
Multiple Transmitter Fault Flag
Unexpected read Data Fault Flag
SBI Parity Fault Flag
9.6.4 SBI SILO DATA REGISTER (SBIS)

The SBI Silo is a history of the state of the indicated SBI signals for the past 16 SBI cycles. The silo is updated every cycle until FAULT is asserted on the SBI or an SBI Silo Comparator match occurs. Each entry in the silo has the following format:

```
| A | I | F | N | ID | TAG | SBI | CNF | SBI TR<15:0> |
```

READ ONLY

- **0:15 SBIF TR**: SBI Transmit/Receive Lines
- **17:16 CNF**: SBI Confirmation Lines
- **21:18 SBI**: SBI bits 21-18 are written with SBI B31-B28 when SBI TAG FIELD specifies command address TAG. Otherwise, M3-M0 are written in this field.
- **24:22 TAG**: SBI TAG
- **29:25 ID**: SBI IDI
- **30 INT**: INTLK
- **31 AFT**: AFT FLT

First Entry after FAULT cleared

9.6.5 SBI SILO COMPARATOR REGISTER (SBISC)

The Silo Comparator allows the SBI to become locked when pre specified conditions are detected. Conditional and unconditional lock modes are provided by the Silo Comparator.
9.6.6 SBI MAINTENANCE REGISTER (SBIMT)

The SBI Maintenance register allows error conditions to be forced for diagnostic purposes.
Privileged Registers 13-May-81 -- Rev 5.2

VAX-11/780 SPECIFIC REGISTERS

<table>
<thead>
<tr>
<th>P</th>
<th>W</th>
<th>U</th>
<th>M</th>
<th>I</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>N</td>
<td>L</td>
<td>MAINT ID</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>X</td>
<td>T</td>
<td>V</td>
<td>I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0:7 MBZ
8 TIM TIME F OUT Force Timeout on Read
9 GØ GØ MAT Group Ø Match
10 G1 G1 MAT Group 1 Match
11 P1 REV SBI P1 Force P1 reversal on SBI
12 DSB DSB SBI CYC Disable SBI Cycles
13 FR1 F G1 REP Force Cache Replacement Group 1
14 FRØ F GØ REP Force Cache Replacement Group Ø
15 FG1 F G1 MISS Force Cache Miss Group 1
16 FGØ MISS F GØ Force Cache Miss Group Ø
20:17 REV REV CACHE PAR FIELD Reverse Cache Parity Field
21 ENI EN SBI INV Enable SBI Invalidate
22 INV INV F SBI Force SBI Write Invalidate to Cache
27:23 MAINT ID Maintenance ID - to force faults and as SILO Comparator
28 MLT MLT F XMIT Force Multiple Transmitter Fault
29 UNX UNEX F RD Force Unexpected Read Data Fault
30 WRT WRT F SEQ Force Write Sequence Fault
31 PØ PØ REV SBI Force PØ Reversal on SBI

9.6.7 SBI ERROR REGISTER (SBIER)
Privileged Registers

VAX-11/780 SPECIFIC REGISTERS

9.6.8 SBI TIMEOUT ADDRESS (SBITA)

This register is a holding register for the Physical Address sent on the SBI. When a timeout occurs on the SBI, this register will latch up with the physical address of the timeout. It is reset by clearing bit 12 of the SBI error register.
Privileged Registers

VAX-11/780 SPECIFIC REGISTERS

3 3 2 2 2
1 0 9 8 7
+---+---+---+---+---+
| M | M | P | | 0 |
| 1 | 0 | C | 0 | |
+---+---+---+---+---+
PHYSICAL ADDRESS <29:2>

READ ONLY

27:0 PHYSICAL ADDRESS <29:2>
28 0
29 PC NO PROT CHK Protection checked reference.
30 M0 Mode 0 reference
31 M1 Mode 1 reference

9.6.9 SBI QUAD CLEAR (SBIQC)

3 3 2
1 0 9
+---+---+---+
| MBZ | PHYSICAL QUADWORD ADDRESS | MBZ |
+---+---+---+

WRITE ONLY
9.7 VAX-11/750 SPECIFIC REGISTERS

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Mnemonic</th>
<th>Number</th>
<th>Type</th>
<th>Scope</th>
<th>Init?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMI Error Register</td>
<td>CMIERR</td>
<td>23</td>
<td>R</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Console Storage Receiver Status</td>
<td>CSRS</td>
<td>28</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Console Storage Receiver Data</td>
<td>CSRD</td>
<td>29</td>
<td>R</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Console Storage Transmit Status</td>
<td>CSTS</td>
<td>30</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Console Storage Transmit Data</td>
<td>CSTD</td>
<td>31</td>
<td>W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Translation Buffer Disable</td>
<td>TBDR</td>
<td>36</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Cache Disable</td>
<td>CADR</td>
<td>37</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Machine Check Error Summary</td>
<td>MCESR</td>
<td>38</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Cache Error</td>
<td>CAER</td>
<td>39</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Accelerator Control/Status</td>
<td>ACCS</td>
<td>40</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Initialize UNIBUS</td>
<td>IORESET</td>
<td>55</td>
<td>W</td>
<td>CPU</td>
<td>--</td>
</tr>
<tr>
<td>Translation Buffer Data</td>
<td>TBDATA</td>
<td>59</td>
<td>R/W</td>
<td>CPU</td>
<td>--</td>
</tr>
</tbody>
</table>

9.7.1 CMI Error Register

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0:3 BER Bus Error
0 Corrected Data Error
1 Lost Error
2 Uncorrectable Data Error
3 Non-existent memory
4 TBHT Hit on last reference
11:8 TBGPR TB Group Error
8 TB Group 0 Data error
9 TB Group 1 Data Error
10 TB Group 0 Tag Error
11 TB Group 1 Tag error
12 RLTO Read Lock Timeout
18:16 SMR Saved Mode Register
17:16 Processor access mode for last reference
18 Virtual=0, Physical=1
20 CMIDIS Disable CMI references
9.7.2 Console Storage Device Registers

The VAX-11/750 accesses the console storage device through four internal registers that are distinct from those used to access the console terminal. The architecture of these registers is similar to that of the console terminal registers.

```
+---------------------+ 8 7 6 5 0
|                    |
|  MBZ               |
|                    |
+---------------------+ 8 7 6 5 0
|                    |
|  R R               |
|  O W               |

Console Storage Receive Status (CSRS)

```

```
+---------------------+ 8 7 0
|                    |
|  0                | DATA  |
+---------------------+ 8 7 0

(read only)

Console Storage Receive Data Buffer (CSRD)

```

```
+---------------------+ 8 7 6 5 1 0
|                    |
|  MBZ               |
|                    |
+---------------------+ 8 7 6 5 1 0
|                    |
|  R R               |
|  O W               |

Console Storage Transmit Status (CSTS)

```

```
+---------------------+ 8 7 0
|                    |
|  0                | DATA  |
+---------------------+ 8 7 0

(write only)

Console Storage Transmit Data Buffer (CSTD)
```
9.7.3 Translation Buffer Group Disable Register (TBDR)

3
1
+--------------------------+--
|                          |   | 4 3 2 1 0
|                          |   | +++++++++
0
|                          |   |           
| Force Miss Group Ø      |   |           
1
| Force Miss Group 1      |   |           
2
| if {<3> EQL 1} then this bit selects group to be replaced |   |
3
| 0 = Random replacement, 1 = Force replacement         |   |

9.7.4 Cache Disable Register (CADR)

3
1
+--------------------------+--
|                          |   | 1 0
|                          |   | +++++++++
0
|                          |   |
| Disable cache            |   |

9.7.5 Machine Check Error Summary Register (MCESR)

3
1
+--------------------------+--
|                          |   | 4 3 2 1 0
|                          |   | +++++++++
0
| Reference was through prefetch logic            |   |
2
| TB parity error                              |   |
3
| Bus error                                    |   |

9.7.6 Cache Error Register (CAER)
Privileged Registers 13-May-81 -- Rev 5.2
VAX-11/750 SPECIFIC REGISTERS

3 1
+-----------------------------
| 4 3 2 1 0
| +++++++
+-----------------------------

Ø                Cache hit
1                Lost error
2                Cache data parity error
3                Cache Tag parity error

9.7.7 Accelerator Control/Status Register

The accelerator control and status register on the VAX-11/750 is a subset of the ACCS on the VAX-11/780.

3 1
+-----------------------------
| 1 1 1
| 6 5 4
| 8 7 0
| +++++++
+-----------------------------

| MBZ |
| E  |
| N  |
| MBZ|
| B  |
| +++++++

W 0
R0

<7:0> TYPE Ø = no accelerator or disabled
       1 = Floating Point Accelerator (FPA)
       Numbers in the range 2-127 are reserved to DIGITAL.
       Numbers in the range 128-255 are reserved to CSS/customers.
<15> ENB Enable FPA.

ACCS<15> always reads as 0. To determine if an FPA is present, write a 1 to ACCS<15> and then read ACCS<Ø>. If there is no FPA, ACCS<Ø> will read as 0.

9.7.8 Initialize UNIBUS (IORESET)

3 1
+-----------------------------
| 1 0
| +++++++
| MBZ |
| +++++++

<Ø> Initialize Unibus
9.7.9 Translation Buffer Data Register (TBDATA)

This internal processor register is used to read and write locations in the TB. On a MPPR to this register, the page table entry for the virtual address in P0BR is read from the TB into the destination. On a MTPR, the source operand is written into the TB as the page table entry for the virtual address in P0BR. The results of an MTPR/MPPR on the register are UNPREDICTABLE if memory management is enabled.
CHAPTER 10
PDP-11 COMPATIBILITY MODE

23-March-81 -- Rev 5.2

10.1 INTRODUCTION

VAX compatibility mode hardware, in conjunction with a compatibility mode software executive (which runs in VAX mode), can emulate the environment provided to user programs on a PDP-11. This environment excludes the following features of normal PDP-11 operation:

1. Privileged instructions such as HALT and RESET.
2. Special instructions such as traps and WAIT.
3. Access to internal processor registers (e.g., PSW and console switch register).
4. Direct access to trap and interrupt vectors.
5. Direct access to I/O devices.
6. Interrupt servicing.
7. Stack overflow protection.
8. Alternate general register sets.
9. Any processor mode other than user (i.e., Kernel and Supervisor modes are not supported) and separate I and D spaces.
10. Floating point instructions.

This specification is based on the behavior of all PDP-11 implementations. Compatibility mode behavior is defined as UNPREDICTABLE where there is a difference between any two PDP-11 implementations.
10.2 COMPATIBILITY MODE USER ENVIRONMENT

10.2.1 General Registers And Addressing Modes

All of the PDP-11 general registers and addressing modes are provided in compatibility mode. Side effects caused by a destination address calculation have no effect on source values (except in JSR), and auto-increment modes in JMP and JSR do not affect the new PC. However, side effects caused by a source address calculation might affect the value of a register used for destination address calculation. All PDP-11 addresses are 16 bits wide. In compatibility mode, a 16-bit PDP-11 address is zero-extended to 32 bits.

10.2.1.1 Register Mode -

The addressing format for register mode is:

```
  5 3 2  0
+-------+
| 0 | Rn |
+-------+
```

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

```
operand = Rn
```

Byte operations, except for MOVB to a register, access the low order byte, i.e. bits <7:0>. The low byte is sign-extended if a register is used as the destination of a MOVB instruction. If the PC is used as the destination of a byte instruction, the result is UNPREDICTABLE.

The assembler notation for register mode is Rn.

10.2.1.2 Register Deferred Mode -

The addressing format for register deferred mode is:

```
  5 3 2  0
+-------+
| 1 | Rn |
+-------+
```

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

```
OA = Rn
operand = (OA)
```
The assembler notation for register deferred mode is (Rn) or @Rn.

10.2.1.3 Autoincrement Mode -

The addressing format for autoincrement mode is:

\[
\begin{array}{cccc}
5 & 3 & 2 & 0 \\
+\cdots\cdots+ \\
| & 2 & | & Rn | \\
+\cdots\cdots+
\end{array}
\]

If Rn denotes PC, immediate data follows the instruction, and the mode is termed immediate mode.

In autoincrement mode addressing, the address of the operand is the contents of register n. After the operand address is determined, the size of the operand in bytes (1 for byte; 2 for word) is added to the contents of register n (except in the case of SP and PC), and the register is replaced by the result. If Rn denotes SP or PC, the register is incremented by 2 and the register is replaced by the result.

\[OA = Rn\]

if n LEQ 5 then \(Rn \leftarrow Rn + \text{size}\) else \(Rn \leftarrow Rn + 2\)

\[\text{operand} = (OA)\]

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

10.2.1.4 Autoincrement Deferred Mode -

The addressing format for autoincrement deferred mode is:

\[
\begin{array}{cccc}
5 & 3 & 2 & 0 \\
+\cdots\cdots+ \\
| & 3 & | & Rn | \\
+\cdots\cdots+
\end{array}
\]

If Rn denotes PC, a 16-bit address follows the instruction, and the mode is termed absolute mode.

In autoincrement deferred mode addressing, the address of the operand is the contents of a word whose address is the contents of register n. After the operand address is determined, 2 is added to the contents of register n, and the register is replaced by the result.
OA = (Rn)

Rn <- Rn + 2

operand = (OA)

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

10.2.1.5 Autodecrement Mode -

The addressing format for autodecrement mode is:

5 3 2 0
+--------+
| 4 | Rn |
+--------+

In autodecrement mode addressing, the size of the operand in bytes (1 for byte; 2 for word) is subtracted from the contents of register n (except in the case of SP and PC), and the register is replaced by the result. If Rn denotes SP or PC, the register is decremented by 2 and the register is replaced by the result. The updated contents of register n is the address of the operand:

if n LEQ 5 then Rn <- Rn - size else Rn <- Rn - 2

OA = Rn

operand = (OA)

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

10.2.1.6 Autodecrement Deferred Mode -

The addressing format for autodecrement deferred mode is:

5 3 2 0
+--------+
| 5 | Rn |
+--------+

In autodecrement deferred mode addressing, 2 is subtracted from the contents of register n, and the register is replaced by the result. The updated contents of register n is the address of the word whose contents is the address of the operand:
Rn <- Rn - 2
OA = (Rn)
operand = (OA)

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

10.2.1.7 Index Mode -
The addressing format for index mode is:

```
  5  3  2  0
+----------+
| 6   Rn   |
+----------+
```

In index mode, the index (contents of the word following the instruction) is added to the contents of register n. The result is the address of the operand:

```
OA = Rn + index
operand = (OA)
```

If Rn denotes PC, the updated contents of the PC is used, and the mode is termed relative mode.

The assembler notation for index mode is index(Rn), where the index value is the word following the instruction.

10.2.1.8 Index Deferred Mode -
The addressing format for index deferred mode is:

```
  5  3  2  0
+----------+
| 7   Rn   |
+----------+
```

In index deferred mode, the index (contents of the word following the instruction) is added to the contents of register n. The result is the address of a word whose contents is the address of the operand:

```
OA = (Rn + index)
operand = (OA)
```
If Rn denotes PC, the updated contents of the PC is used, and the mode is termed relative deferred mode.

The assembler notation for index deferred mode is @index(Rn), where the index value is the word following the instruction.

10.2.2 The Stack

General register R6 is used as the stack pointer by certain instructions, as in the PDP-11. It is not, however, used by the hardware for any exceptions or interrupts. There is also no stack overflow protection in compatibility mode.

10.2.3 Processor Status Word

PDP-11 compatibility mode uses a subset of the full PDP-11 Processor Status Word. The format of the compatibility mode PSW is:

```
  1
  5 5 4 3 2 1 0
+-------------------+----------+
| T|N|Z|V|C|       +----------+
+-------------------+----------+
```

When an RTI or RTT instruction is executed, bits 15 through 5 in the saved PSW on the stack are ignored.
10.2.4 Instructions

Table 10.1 lists the instructions provided in compatibility mode.

<table>
<thead>
<tr>
<th>Opcode (octal)</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>000002</td>
<td>RTI</td>
</tr>
<tr>
<td>000006</td>
<td>RTT</td>
</tr>
<tr>
<td>0001DD</td>
<td>JMP</td>
</tr>
<tr>
<td>00020R</td>
<td>RTS</td>
</tr>
<tr>
<td>000240-000277</td>
<td>Condition codes</td>
</tr>
<tr>
<td>0003DD</td>
<td>SWAB</td>
</tr>
<tr>
<td>000400-003777</td>
<td>Branches</td>
</tr>
<tr>
<td>100000-103777</td>
<td>Branches</td>
</tr>
<tr>
<td>004RDD</td>
<td>JSR</td>
</tr>
<tr>
<td>.050DD</td>
<td>CLR(B)</td>
</tr>
<tr>
<td>.051DD</td>
<td>COM(B)</td>
</tr>
<tr>
<td>.052DD</td>
<td>INC(B)</td>
</tr>
<tr>
<td>.053DD</td>
<td>DEC(B)</td>
</tr>
<tr>
<td>.054DD</td>
<td>NEG(B)</td>
</tr>
<tr>
<td>.055DD</td>
<td>ADC(B)</td>
</tr>
<tr>
<td>.056DD</td>
<td>SBC(B)</td>
</tr>
<tr>
<td>.057SS</td>
<td>TST(B)</td>
</tr>
<tr>
<td>.060DD</td>
<td>ROR(B)</td>
</tr>
<tr>
<td>.061DD</td>
<td>ROL(B)</td>
</tr>
<tr>
<td>.062DD</td>
<td>ASR(B)</td>
</tr>
<tr>
<td>.063DD</td>
<td>ASL(B)</td>
</tr>
<tr>
<td>0065SS</td>
<td>MFPI*</td>
</tr>
<tr>
<td>0066DD</td>
<td>MTPD*</td>
</tr>
<tr>
<td>1065SS</td>
<td>MFPI*</td>
</tr>
<tr>
<td>1066DD</td>
<td>MTPD*</td>
</tr>
<tr>
<td>0067DD</td>
<td>SXT</td>
</tr>
<tr>
<td>070RSS</td>
<td>MUL</td>
</tr>
<tr>
<td>071RSS</td>
<td>DIV</td>
</tr>
<tr>
<td>072RSS</td>
<td>ASH</td>
</tr>
<tr>
<td>073RSS</td>
<td>ASHC</td>
</tr>
<tr>
<td>074RDD</td>
<td>XOR</td>
</tr>
<tr>
<td>077RNN</td>
<td>SOB</td>
</tr>
<tr>
<td>.1SSDD</td>
<td>MOV(B)</td>
</tr>
<tr>
<td>.2SSSS</td>
<td>CMP(B)</td>
</tr>
<tr>
<td>.3SSSS</td>
<td>BIT(B)</td>
</tr>
<tr>
<td>.4SSDD</td>
<td>BIC(B)</td>
</tr>
<tr>
<td>.5SSDD</td>
<td>BIS(B)</td>
</tr>
<tr>
<td>06SSDD</td>
<td>ADD</td>
</tr>
<tr>
<td>16SSDD</td>
<td>SUB</td>
</tr>
</tbody>
</table>

R = register specifier
SS = source operand specifier
DD = destination operand specifier
. = 0 for word operations and 1 for byte operations
* These instructions execute exactly as they would on a PDP-11 in user mode with Instruction and Data space overmapped. More specifically, they ignore the previous access level and act like PUSH and POP instructions referencing the current stack.

Table 10.2 lists the trap instructions that cause the machine to fault to VAX mode, where either the complete trap may be serviced, or where the instruction may be simulated.

### TABLE 10.2
Compatibility Mode Trap Instructions

<table>
<thead>
<tr>
<th>Opcode (octal)</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>000003</td>
<td>BPT</td>
</tr>
<tr>
<td>000004</td>
<td>IOT</td>
</tr>
<tr>
<td>104000-104377</td>
<td>EMT</td>
</tr>
<tr>
<td>104400-104777</td>
<td>TRAP</td>
</tr>
</tbody>
</table>

The instructions listed in Table 10.3 and all other opcodes not listed in Tables 10.1 or 10.2 are considered reserved instructions in compatibility mode, and fault to VAX mode. See Section 10.5.

### TABLE 10.3
Compatibility Mode Reserved Instructions

<table>
<thead>
<tr>
<th>Opcode (octal)</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>HALT</td>
</tr>
<tr>
<td>000001</td>
<td>WAIT</td>
</tr>
<tr>
<td>000005</td>
<td>RESET</td>
</tr>
<tr>
<td>000007</td>
<td>MFPT</td>
</tr>
<tr>
<td>00023N</td>
<td>SPL</td>
</tr>
<tr>
<td>0064NN</td>
<td>MARK</td>
</tr>
<tr>
<td>0070DD</td>
<td>CSM</td>
</tr>
<tr>
<td>07500R</td>
<td>FADD--FIS</td>
</tr>
<tr>
<td>07501R</td>
<td>FSUB--FIS</td>
</tr>
<tr>
<td>07502R</td>
<td>FMUL--FIS</td>
</tr>
<tr>
<td>07503R</td>
<td>FDIV--FIS</td>
</tr>
<tr>
<td>076XXX</td>
<td>Extended Instructions</td>
</tr>
<tr>
<td>10645S</td>
<td>MTPS</td>
</tr>
<tr>
<td>1067DD</td>
<td>MFFPS</td>
</tr>
<tr>
<td>17XXX</td>
<td>FPP1 Floating Point</td>
</tr>
</tbody>
</table>

Note that no floating point instructions are included in compatibility mode.
10.2.4.1 Single Operand Instructions -

Arithmetic and Logical:

<table>
<thead>
<tr>
<th>CLR</th>
<th>DEC</th>
<th>INC</th>
<th>NEG</th>
<th>TST</th>
<th>COM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>DEC</td>
<td>INC</td>
<td>NEG</td>
<td>TST</td>
<td>COM</td>
</tr>
</tbody>
</table>

Shifts:

<table>
<thead>
<tr>
<th>ASR</th>
<th>ASL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR</td>
<td>ASL</td>
</tr>
</tbody>
</table>

Multiprecision:

<table>
<thead>
<tr>
<th>ADC</th>
<th>SBC</th>
<th>SXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>SBC</td>
<td>SXT</td>
</tr>
</tbody>
</table>

Rotates:

<table>
<thead>
<tr>
<th>ROL</th>
<th>ROR</th>
<th>SWAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROL</td>
<td>ROR</td>
<td>SWAB</td>
</tr>
<tr>
<td>ROL</td>
<td>ROR</td>
<td>SWAB</td>
</tr>
</tbody>
</table>
CLR  Clear

Format:

```
  1
  5 6 5 0
  +---------------------+
  | Opcode | dst.wx |
  +---------------------+
```

Operation:

```
dst <- 0;
```

Condition Codes:

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

Exceptions:

```
none
```

Opcodes (octal):

```
0050  CLR  Clear Word
1050  CLRB Clear Byte
```

Description:

The destination operand is replaced by zero.
DEC     Decrement

Format:

\[
\begin{array}{ccc}
1 & 6 & 5 & 0 \\
| \hline\hline
| \text{Opcode} & \text{dst.mx} \\
\end{array}
\]

Operation:

\[
dst \leftarrow dst - 1;
\]

Condition Codes:

\[
N \leftarrow dst \text{ LSS } 0; \\
Z \leftarrow dst \text{ EQL } 0; \\
V \leftarrow \{\text{integer overflow}\}; \\
C \leftarrow C;
\]

Exceptions:

none

Opcodes (octal):

\[
\begin{array}{ccc}
0053 & \text{DEC} & \text{Decrement Word} \\
1053 & \text{DECB} & \text{Decrement Byte} \\
\end{array}
\]

Description:

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

Note:

Integer overflow occurs if the largest negative integer is decremented. On overflow, the destination operand is replaced by the largest positive integer.
INC     Increment

Format:

```
 5   5   0
+-------------------+
   |    Opcode     | dst.mx |
+-------------------+
```

Operation:

```
dst <- dst + 1;
```

Condition Codes:

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

Exceptions:

```
none
```

Opcodes (octal):

```
0052   INC      Increment Word
1052   INCB     Increment Byte
```

Description:

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

Note:

Integer overflow occurs if the largest positive integer is incremented. On overflow, the destination operand is replaced by the largest negative integer.
NEG     Negate

Format:

```
  1
  5  6  5  0
```

```
+----------------------------------+
| Opcode   | dst.mx   |                   |
+----------------------------------+
```

Operation:

dst <= -dst;

Condition Codes:

N <= dst LSS 0;
Z <= dst EQL 0;
V <= dst EQL most negative integer;
C <= dst NEQ 0;

Exceptions:

none

Opcodes (octal):

<table>
<thead>
<tr>
<th>Octal</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0054</td>
<td>NEG</td>
<td>Negate Word</td>
</tr>
<tr>
<td>1054</td>
<td>NEGB</td>
<td>Negate Byte</td>
</tr>
</tbody>
</table>

Description:

The destination operand is negated (2's complement) and the destination operand is replaced by the result.

Note:

Integer overflow occurs if the operand is the most negative integer (which has no positive counterpart). On overflow, the destination operand is replaced by itself.
TST    Test

Format:

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 6 5 0</td>
</tr>
<tr>
<td>-----------------+-----------------</td>
</tr>
<tr>
<td>Opcode</td>
</tr>
<tr>
<td>-----------------+-----------------</td>
</tr>
</tbody>
</table>

Operation:

src - 0;

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

<table>
<thead>
<tr>
<th>0057</th>
<th>TST</th>
<th>Test Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>1057</td>
<td>TSTB</td>
<td>Test Byte</td>
</tr>
</tbody>
</table>

Description:

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

Format:

```
  1
  5
  6 5 0
  +---------------------+
  | Opcode   | dst.mx |
  +---------------------+
```

Operation:

\[
dst \leftarrow \text{NOT} \ dst;
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow \text{dst LSS 0}; \\
Z & \leftarrow \text{dst EQL 0}; \\
V & \leftarrow 0; \\
C & \leftarrow 1;
\end{align*}
\]

Exceptions:

none

Opcodes (octal):

<table>
<thead>
<tr>
<th>Code</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0051</td>
<td>COM</td>
<td>Complement Word</td>
</tr>
<tr>
<td>1051</td>
<td>COMB</td>
<td>Complement Byte</td>
</tr>
</tbody>
</table>

Description:

The destination operand is complemented (1's complement) and the destination operand is replaced by the result.
ASR  Arithmetic Shift Right

Format:

\[
\begin{array}{cccc}
\alpha & b & 6 & 5 & \gamma,
\end{array}
\]

Operation:

\[
dst \leftarrow dst \text{ shifted one place to the right;}
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow dst \text{ LSS } 0; \\
z & \leftarrow dst \text{ EQL } 0; \\
v & \leftarrow \text{bit shifted out} \text{ XOR } (dst \text{ LSS } 0); \\
c & \leftarrow \text{bit shifted out;}
\end{align*}
\]

Exceptions:

none

Opcodes (octal):

\[
\begin{align*}
0062 & \quad ASR & \quad \text{Arithmetic Shift Right Word} \\
1062 & \quad ASRB & \quad \text{Arithmetic Shift Right Byte}
\end{align*}
\]

Description:

The destination operand is arithmetically shifted right by one bit and the destination operand is replaced by the result.

Notes:

1. The sign bit of the destination operand is replicated in shifts to the right. The condition code C bit stores the bit shifted out.

2. If the PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.
ASL Arithmetic Shift Left

Format:

```
1 6 5 0
+--------+
| Opcode | dst.mx |
+--------+
```

Operation:

dst <- dst shifted one place to the left;

Condition Codes:

N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- bit shifted out;

Exceptions:

none

Opcodes (octal):

<table>
<thead>
<tr>
<th>0063</th>
<th>ASL</th>
<th>Arithmetic Shift Left Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>1063</td>
<td>ASLB</td>
<td>Arithmetic Shift Left Byte</td>
</tr>
</tbody>
</table>

Description:

The destination operand is arithmetically shifted left by one bit and the destination operand is replaced by the result.

Notes:

1. The least significant bit is filled with zero in shifts to the left. The condition code C bit stores the bit shifted out.
2. Integer overflow occurs if the destination changes sign due to the shift.
ADC      Add Carry

Format:

```
  1
  5
+----------------+------------------------+
| Opcode | dst.mx |
+----------------+------------------------+
```

Operation:

```plaintext
dst <- dst + C;
```

Condition Codes:

- N <- dst LSS 0;
- Z <- dst EQ 0;
- V <- {integer overflow};
- C <- {carry from most significant bit};

Exceptions:

- none

Opcodes (octal):

- `0055` ADC Add Carry to Word
- `1055` ADCB Add Carry to Byte

Description:

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

Note:

Integer overflow occurs if the most positive integer is incremented. On overflow, the result is the most negative integer.
SBC    Subtract Carry

Format:

```
   1
   5
---+---+---
 |   |   |
 +---+---+
```

Operation:

```
dst <- dst - C;
```

Condition Codes:

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

Exceptions:

```
none
```

Opcodes (octal):

```
0056   SBC    Subtract Carry from Word
1056   SBCB   Subtract Carry from Byte
```

Description:

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

Note:

Integer overflow occurs if the most negative integer is decremented. On overflow, the result is the most positive integer.
SXT      Sign Extend Word

Format:

\[
\begin{array}{cccc}
  & 1 & 5 & 6 & 5 & 0 \\
+-----------------------------+ \\
| \text{Opcode} & \text{dst.ww} | \\
+-----------------------------+
\end{array}
\]

Operation:

\[
\text{if } N \text{ EQL } 1 \text{ then } \text{dst} <- -1 \text{ else } \text{dst} <- 0;
\]

Condition Codes:

\[
\begin{align*}
N & <- \text{dst LSS } 0; \ N & <- \ N \\
Z & <- \text{dst EQL } 0; \\
V & <- 0; \\
C & <- C;
\end{align*}
\]

Exceptions:

none

Opcodes (octal):

\[
\begin{align*}
0067 & \quad \text{SXT Sign Extend}
\end{align*}
\]

Description:

If the condition code N bit is set then the destination operand is replaced by -1; otherwise the destination operand is cleared.

Note:

If the PC is used as the destination operand, the results and the next instruction executed are UNPREDICTABLE.
ROL    Rotate Left

Format:

```
  6 5 0
 5 +----------------+----------------+
 |   Opcode      |   dst.mx     |
```

Operation:

dst'C <- dst'C rotated left;

Condition Codes:

- \( N \leftarrow \text{dst LSS 0}; \)
- \( Z \leftarrow \text{dst EQL 0}; \)
- \( V \leftarrow \{\text{integer overflow}\}; \)
- \( C \leftarrow \{\text{bit rotated out of dst}\}; \)

Exceptions:

- none

Opcodes (octal):

- \( 0061 \quad \text{ROL} \quad \text{Rotate Left Word} \)
- \( 1061 \quad \text{ROLB} \quad \text{Rotate Left Byte} \)

Description:

The condition code C bit and the destination operand are rotated left by one bit position; i.e., the C bit gets the most significant bit of the destination operand, the destination is replaced by the destination shifted left by one bit with the initial C bit filling the least significant bit.

Notes:

1. The rotate instructions operate on the destination operand and the condition code C bit taken as a circular datum.

2. Integer overflow occurs if the destination changes sign due to the rotate.
ROR    Rotate Right

Format:

\[
\begin{array}{ccc}
& 6 & 5 & 0 \\
\hline
| Opcode | dst.mx |
\end{array}
\]

Operation:

\[
\text{dst}'C \leftarrow \text{dst}'C \text{ rotated right};
\]

Condition Codes:

\[
\begin{align*}
N \leftarrow & \text{dst LSS 0;} \\
z \leftarrow & \text{dst EQL 0;} \\
v \leftarrow & \{\text{C bit changed due to rotate}; \\
c \leftarrow & \{\text{bit rotated out of dst}; \\
\end{align*}
\]

Exceptions:

none

Opcodes (octal):

\[
\begin{align*}
0060 & \quad \text{ROR} \quad \text{Rotate Right Word} \\
1060 & \quad \text{RORB} \quad \text{Rotate Right Byte}
\end{align*}
\]

Description:

The condition code C bit and the destination operand are rotated right by one bit position; i.e. the C bit gets the least significant bit of the destination operand, the destination is replaced by the destination shifted right by one bit with the initial C bit filling the most significant bit.

Note:

The rotate instructions operate on the destination operand and the condition code C bit taken as a circular datum.
SWAB  Swap Bytes

Format:

```
+---+---+---+
| 5 | 6 5| 0 |
+---+---+---+
```

```
Opcode | dst.mw |
+-------+--------+
```

Operation:

```
dst <- dst<7:0>','dst<15:8>;
```

Condition Codes:

```
N <- dst<7:0> LSS 0;
Z <- dst<7:0> EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes (octal):

```
0003  SWAB  Swap Bytes
```

Description:

The high and low bytes of the destination word operand are swapped.

Note:

If the PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.
10.2.4.2 Double Operand Instructions -

Arithmetic and Logical:

<table>
<thead>
<tr>
<th>MOV</th>
<th>ADD</th>
<th>SUB</th>
<th>CMP</th>
<th>MUL</th>
<th>DIV</th>
<th>XOR</th>
<th>BIS</th>
<th>BIC</th>
<th>BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVB</td>
<td>CMPB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shift:

| ASH | ASHC |

If a register that is used in the source operand specifier in autoincrement or autodecrement modes is also used in the destination (or source 2) operand specifier, the updated value of the register is used to evaluate the destination specifier. Side effects caused by a destination address calculation have no effect on source values.
MOV     Move

Format:

```
      1  1 1  6 5  0
  5  2 1  +---------------------+
  |Opcode | src.rx | dst.wx |
  +---------------------+
```

Operation:

\[ \text{dst} \leftarrow \text{src}; \]

Condition Codes:

\[
\begin{align*}
\text{N} & \leftarrow \text{dst \text{LSS} 0}; \\
\text{Z} & \leftarrow \text{dst \text{EQL} 0}; \\
\text{V} & \leftarrow 0; \\
\text{C} & \leftarrow \text{C};
\end{align*}
\]

Exceptions:

none

Opcodes (octal):

\[
\begin{align*}
01 & \text{ MOV } \quad \text{Move Word} \\
11 & \text{ MOVB } \quad \text{Move Byte}
\end{align*}
\]

Description:

The destination operand is replaced by the source operand.

Note:

The low byte is sign-extended on a MOVB to a register; i.e. bits \( <15:8> \) of the destination register are replaced by bit \( <7> \) of the source operand.
ADD     Add

Format:

\[
\begin{array}{cccc}
1 & 1 & 1 & 6 & 5 & 0 \\
5 & 2 & 1 & & & \\
\hline
\text{Opcode} & \text{src.rw} & \text{dst.mw} \\
\hline
\end{array}
\]

Operation:

dst <- dst + src;

Condition Codes:

\[
\begin{align*}
N & \leftarrow \text{dst LSS 0}; \\
Z & \leftarrow \text{dst EQL 0}; \\
V & \leftarrow \text{[integer overflow]}; \\
C & \leftarrow \text{[carry from most significant digit]};
\end{align*}
\]

Exceptions:

none

Opcodes (octal):

\[
06 \quad \text{ADD} \quad \text{Add Word}
\]

Description:

The source operand is added to the destination operand and the destination operand is replaced by the result.

Note:

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

Format:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>src.rw</th>
<th>dst.mw</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>+-------+-------+-------</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:

dst <- dst - src;

Condition Codes:

N <- dst LSS 0;
Z <- dst EQL 0;
V <- [integer overflow];
C <- [borrow into most significant digit];

Exceptions:

none

Opcodes (octal):

15    SUB    Subtract Word

Description:

The source operand is subtracted from the destination operand and the destination operand is replaced by the result.

Note:

Integer overflow occurs if the input operands are of different signs and the result has the sign of the source. On overflow, the destination operand is replaced by the low order bits of the true result.
CMP    Compare

Format:

\[
\begin{array}{c|c|c|c|c|c}
1 & 1 & 1 & 6 & 5 & 0 \\
\hline
5 & 2 & 1 & \_ & \_ & \_ \\
\hline
\end{array}
\]

Operation:

\[
tmp \leftarrow src1 - src2;
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow \text{tmp LSS 0}; \\
Z & \leftarrow \text{tmp EQL 0}; \\
v & \leftarrow \{\text{integer overflow}\}; \\
c & \leftarrow \{\text{borrow into most significant digit}\};
\end{align*}
\]

Exceptions:

none

Opcodes (octal):

\[
\begin{array}{c|c}
02 & \text{CMP} \quad \text{Compare Word} \\
12 & \text{CMPB} \quad \text{Compare Byte} \\
\end{array}
\]

Description:

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

Note:

Integer overflow occurs if the operands are of different sign and the result of the subtraction \((\text{src1} - \text{src2})\) has the same sign as the source 2 operand.
MUL       Multiply

Format:

```
  5  9 8 6 5 0
+-------------------+---+
| Opcode | reg | src.rw |
+-------------------+---+
```

Operation:

```
tmp<31:0> <- Rn * src;
Rn <- tmp<31:15>;
R[n OR 1] <- tmp<15:0>;
```

Condition Codes:

```
N <- tmp LSS 0;
Z <- tmp EQL 0;
V <- 0;
C <- {result unrepresentable in 16 bits};
```

Exceptions:

none

Opcodes (octal):

```
070       MUL       Multiply Word
```

Description:

The destination register is multiplied by the source operand. The most significant 16 bits of the 32-bit product are stored in register Rn. Then the least significant 16 bits are stored in R[n OR 1]. The condition codes are set based on the 32-bit result.

Note:

1. The C bit is set if the result of the multiplication cannot be represented in 16 bits; i.e. the 32-bit product is less than $-2^{15}$ or greater than or equal to $2^{15}$.

2. If an odd numbered register is used as the destination, the low order sixteen bits are stored as the result.

3. If R6 or PC is used as the destination, the next instruction executed and the result are UNPREDICTABLE.
DIV Divide

Format:

5 9 8 6 5 0
-----------------------------
| Opcode | reg | src.rw |
-----------------------------

Operation:

tmp <- Rn'R[n OR 1]
Rn <- tmp / src;
R[n OR 1] <- REM(tmp, src);

Condition Codes:

N <- Rn LSS 0; !UNPREDICTABLE if V is set
Z <- Rn EQL 0; !UNPREDICTABLE if V is set
V <- {src EQL 0} OR {integer overflow};
C <- {src EQL 0};

Exceptions:

none

Opcodes (octal):

071 DIV Divide

Description:

If the source operand is not zero, the 32-bit integer in Rn'R[n OR 1] is
divided by the source operand. The quotient is stored in Rn, and the
remainder is stored in R[n OR 1]. The remainder has the same sign as
the dividend. If the source operand is zero, the instruction terminates
without modifying the destination registers.

Notes:

1. Integer overflow occurs if the quotient is less than -2**15 or
greater than or equal to 2**15. On integer overflow, the
contents of the destination registers are UNPREDICTABLE.

2. If an odd register or R6 is used as the destination, the
results are UNPREDICTABLE. Furthermore, if R6 or PC is used as
the destination, the next instruction executed is
UNPREDICTABLE.
XOR     Exclusive OR

Format:

```
 1
 5 9 8 6 5 0
```

```
| Opcode | reg | dst.mw |
+--------+-----+-------+
```

Operation:

```
dst <- Rn XOR dst;
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

```
074    XOR     Exclusive OR Word
```

Description:

The source register is XORed with the destination operand and the destination operand is replaced by the result.
BIS       Bit Set

Format:

\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
5 & 2 & 1 & 6 \\
5 & 5 & 0 & 0 \\
\end{array}
\]

\[\text{Opcode} \mid \text{src.rx} \mid \text{dst.mx} \mid \text{dst.mx} \mid\]

Operation:

dst <- dst OR src;

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

<table>
<thead>
<tr>
<th>05</th>
<th>BIS</th>
<th>Bit Set Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>BISB</td>
<td>Bit Set Byte</td>
</tr>
</tbody>
</table>

Description:

The source operand is ORed with the destination operand and the destination operand is replaced by the result.
BIC  Bit Clear

Format:

\[
\begin{array}{cccc}
1 & 1 & 1 & 0 \\
5 & 2 & 1 & 6 & 5 \\
\hline
\text{Opcode} & \text{src.rx} & \text{dst.mx} \\
\hline
\end{array}
\]

Operation:

\[
dst \leftarrow dst \text{ AND} \{\text{NOT src}\};
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow dst \text{ LSS } 0; \\
Z & \leftarrow dst \text{ EQL } 0; \\
V & \leftarrow 0; \\
C & \leftarrow C;
\end{align*}
\]

Exceptions:

none

Opcodes (octal):

\[
\begin{array}{cc}
04 & \text{BIC} \\
14 & \text{BICB}
\end{array}
\]

Description:

The destination operand is ANDed with the 1's complement of the source operand and the destination operand is replaced by the result.
BIT     Bit Test

Format:

```
  1  1 1
  5  2 1  6 5  0
+-------------------------+
| Opcode | src1.rx | src2.rx |
+-------------------------+
```

Operation:

```
tmp <- src1 AND src2;
```

Condition Codes:

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

Exceptions:

```
one
```

Opcodes (octal):

```
  03     BIT     Bit Test Word
  13     BITB    Bit Test Byte
```

Description:

The source 1 operand is ANDed with the source 2 operand. The only action is to set the condition codes.
ASH       Arithmetic Shift

Format:

\[
\begin{array}{cccccc}
1 & 5 & 9 & 8 & 6 & 5 & 0 \\
+---------------------+---------------------+
| Opcode | reg | src.rw |
+---------------------+---------------------+
\end{array}
\]

Operation:

Rn \leftarrow \text{Rn shifted src\langle5:0\rangle bits;}

Condition Codes:

N \leftarrow \text{Rn LSS 0;}
Z \leftarrow \text{Rn EQL 0;}
V \leftarrow \text{if src\langle5:0\rangle EQL 0 then 0 else \{integer overflow\};}
C \leftarrow \text{if src\langle5:0\rangle EQL 0 then 0 else \{last bit shifted out\};}

Exceptions:

none

Opcodes (octal):

\[072\] ASH Arithmetic Shift

Description:

The specified register is arithmetically shifted by the number of bits specified by the count operand (bits \langle5:0\rangle of the source operand) and the register is replaced by the result. The count ranges from -32 to +31. A negative count signifies a right shift. A positive count signifies a left shift. A zero count implies no shift; but condition codes are affected.

Notes:

1. The sign bit of Rn is replicated in shifts to the right. The least significant bit is filled with zero in shifts to the left. The C bit stores the last bit shifted out.

2. Integer overflow occurs on a left shift if any bit shifted into the sign position differs from the initial sign bit of the register.

3. If the PC is used as the destination operand the result and the next instruction executed are UNPREDICTABLE.
ASHC  Arithmetic Shift Combined

Format:

```
  5 9 8 6 5 0
+---------------+-
| Opcode | reg | src.rw |
+---------------+-
```

Operation:

```
tmp <- Rn'R[n OR 1];
tmp <- tmp shifted src<5:0> bits;
Rn <- tmp<31:16>;
R[n OR 1] <- tmp<15:0>;
```

Condition Codes:

```
N <- tmp LSS 0;
Z <- tmp EQL 0;
V <- if src<5:0> EQL 0 then 0 else {integer overflow};
C <- if src<5:0> EQL 0 then 0 else {last bit shifted out};
```

Exceptions:

```
none
```

Opcodes (octal):

```
073  ASHC  Arithmetic Shift Combined
```

Description:

The contents of the specified register, Rn, and the register R[n OR 1] are treated as a single 32-bit operand and are shifted by the number of bits specified by the count operand (bits <5:0> of the source operand) and the registers are replaced by the result. First, bits <31:16> of the result are stored in register Rn. Then, bits <15:0> of the result are stored in register R[n OR 1]. The count ranges from -32 to +31. A negative count signifies a right shift. A positive count signifies a left shift. A zero count implies no shift; but condition codes are affected. Condition codes are always set on the 32-bit result.

Notes:

1. The sign bit of Rn is replicated in shifts to the right. The least significant bit is filled with zero in shifts to the left. The C bit stores the last bit shifted out.

2. Integer overflow occurs on a left shift if any bit shifted into the sign position differs from the initial sign bit of the 32-bit operand.
3. If the SP or PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.
10.2.4.3 Branch Instructions -

<table>
<thead>
<tr>
<th>BR</th>
<th>BNE</th>
<th>BPL</th>
<th>BVC</th>
<th>BCC</th>
<th>BGE</th>
<th>BGT</th>
<th>BHI</th>
<th>RHIS</th>
<th>SOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>BMI</td>
<td>BVS</td>
<td>BCS</td>
<td>BLT</td>
<td>BLE</td>
<td>BLOS</td>
<td>BLO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BR     Branch

Format:

```
  1
 5     8    7     0
     +---------------------+
     | opcode   | displ.bb |
     +---------------------+
```

Operation:

$$PC \leftarrow PC + \text{SEXT}(2 \cdot \text{displ})$$

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

`0004` BR     Branch

Description:

Twice the sign-extended displacement is added to the PC and the PC is replaced by the result.
B    Branch on (condition)

Format:

<table>
<thead>
<tr>
<th>1</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>displ.bb</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:

if condition then PC <- PC + SEXT(2*disp);

Condition Codes:

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

Exceptions:

none

Opcodes (octal):           Condition

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0014</td>
<td>BEQ</td>
<td>Z EQL 1</td>
<td>Branch on Equal</td>
</tr>
<tr>
<td>0010</td>
<td>BNE</td>
<td>Z EQL 0</td>
<td>Branch Not Equal</td>
</tr>
<tr>
<td>1004</td>
<td>BMI</td>
<td>N EQL 1</td>
<td>Branch on Minus</td>
</tr>
<tr>
<td>1000</td>
<td>BPL</td>
<td>N EQL 0</td>
<td>Branch on Plus</td>
</tr>
<tr>
<td>1034</td>
<td>BCS,</td>
<td>C EQL 1</td>
<td>Branch on Carry Set,</td>
</tr>
<tr>
<td></td>
<td>BLO</td>
<td></td>
<td>Branch on Lower</td>
</tr>
<tr>
<td>1030</td>
<td>BCC,</td>
<td>C EQL 0</td>
<td>Branch on Carry Clear,</td>
</tr>
<tr>
<td></td>
<td>BHIS</td>
<td></td>
<td>Branch on Higher or Same</td>
</tr>
<tr>
<td>1024</td>
<td>BVS</td>
<td>V EQL 1</td>
<td>Branch on Overflow Set</td>
</tr>
<tr>
<td>1020</td>
<td>BVC</td>
<td>V EQL 0</td>
<td>Branch on Overflow Clear</td>
</tr>
<tr>
<td>0024</td>
<td>BLT</td>
<td>{N XOR V} EQL 1</td>
<td>Branch on Less Than</td>
</tr>
<tr>
<td>0020</td>
<td>BGE</td>
<td>{N XOR V} EQL 0</td>
<td>Branch on Greater Than or Equal</td>
</tr>
<tr>
<td>0034</td>
<td>BLE</td>
<td>{Z OR {N XOR V}}</td>
<td>EQL 1</td>
</tr>
<tr>
<td>0030</td>
<td>BGT</td>
<td>{Z OR {N XOR V}}</td>
<td>EQL 0</td>
</tr>
<tr>
<td>1010</td>
<td>BHI</td>
<td>{C OR Z} EQL 0</td>
<td>Branch on Higher</td>
</tr>
<tr>
<td>1014</td>
<td>BLOS</td>
<td>{C OR Z} EQL 1</td>
<td>Branch on Lower or Same</td>
</tr>
</tbody>
</table>

Description:

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

Format:

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>9 8 6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>reg</td>
<td>displ.b5</td>
<td></td>
</tr>
</tbody>
</table>

Operation:

\[ Rn \leftarrow Rn - 1; \]
if \( Rn \neq 0 \) then \( PC \leftarrow PC - \text{ZEXT}(2 \times \text{displ}) \);

Condition Codes:

\[ N \leftarrow N; \]
\[ Z \leftarrow Z; \]
\[ V \leftarrow V; \]
\[ C \leftarrow C; \]

Exceptions:

none

Opcodes (octal):

\[ 077 \quad \text{SOB} \quad \text{Subtract One and Branch} \]

Description:

One is subtracted from the specified register and the register is replaced by the result. If the register is not equal to zero, twice the zero-extended displacement is subtracted from the PC and the PC is replaced by the result.

Notes:

1. If the PC is specified as the register, the results and the next instruction executed are UNPREDICTABLE.

2. The 6-bit displacement operand is contained in bits \( <5:0> \) of the instruction.
10.2.4.4 Jump And Subroutine Instructions -

JMP  JSR
     RTS
JMP      Jump

Format:

<table>
<thead>
<tr>
<th>1</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>dst.aw</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:

PC <- dst;

Condition Codes:

\[
\begin{align*}
N & \leftarrow N; \\
Z & \leftarrow Z; \\
V & \leftarrow V; \\
C & \leftarrow C;
\end{align*}
\]

Exceptions:

compatibility mode illegal instruction

Opcodes (octal):

\[0001 \quad \text{JMP} \quad \text{Jump}\]

Description:

The PC is replaced by the destination operand.

Note:

A compatibility mode illegal instruction fault occurs if destination mode 0 is used.
JSR  Jump to Subroutine

Format:

<table>
<thead>
<tr>
<th>5 9 8 6 5 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
</tr>
</tbody>
</table>

Operation:

tmp <- dst;
-(SP) <- Rn;  !value of Rn affected by dst specifier evaluation
Rn <- PC;
PC <- tmp;

Condition Codes:

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

Exceptions:

compatibility mode illegal instruction

Opcodes (octal):

004  JSR  Jump to Subroutine

Description:

The source register is pushed on the stack and the source register is replaced by the PC. The PC is replaced by the destination operand.

Notes:

1. A compatibility mode illegal instruction fault occurs if destination mode 0 is used.

2. If the destination uses the same register as the source in the autoincrement or autodecrement addressing modes, the updated contents of the register are pushed on the stack.
RTS       Return from Subroutine

Format:

```
  1
3 2 0
+--------+
| Opcode  | reg |
+--------+
```

Operation:

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

Condition Codes:

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

Exceptions:

```
one
```

Opcodes (octal):

```
00020   RTS       Return from subroutine
```

Description:

The PC is replaced by the destination register. The destination register is replaced by a word popped from the stack.
10.2.4.5 Return From Interrupts And Traps -

RTI RTT
RTI    Return from Interrupt
RTT    Return from Trap

Format:

1
5
+-----------------------+ 0
|                        | Opcode
|                        |
+-----------------------+

Operation:

PC <= (SP)+;
PSW<4:0> <= [(SP)+]<4:0>;

Condition Codes:

N <= saved PSW<3>;
Z <= saved PSW<2>;
V <= saved PSW<1>;
C <= saved PSW<0>;

Exceptions:

none

Opcodes (octal):

000002  RTI    Return from Interrupt
000006  RTT    Return from Trap

Description:

The PC is replaced by the first word popped from the stack. The low 5 bits of the PSW are replaced by the corresponding bits of the second word popped from the stack.

Notes:

1. In compatibility mode, the RTI and RTT instructions ignore the high 11 bits of the PSW popped from the stack.

2. In compatibility mode, the RTI and RTT instructions are identical.
10.2.4.6 Miscellaneous -
MTPI  MTPD  SCC
MFPI  MFPD  CCC
MTP        Move To Previous Space

Format:

\[
\begin{array}{cccc}
1 & 5 & 6 & 5 & 0 \\
+---------------------------------+\
| Opcode | dst\_ww | \\
+---------------------------------+
\end{array}
\]

Operation:

\[
dst \leftarrow (SP)+;
\]

Condition Codes:

\[
\begin{align*}
N & \leftarrow dst\_LSS 0; \\
Z & \leftarrow dst\_EQL 0; \\
V & \leftarrow 0; \\
C & \leftarrow C;
\end{align*}
\]

Exceptions:

none

Opcodes (octal):

\[
\begin{align*}
0066 & \quad \text{MTPI} \quad \text{Move To Previous Instruction Space} \\
1066 & \quad \text{MTPD} \quad \text{Move To Previous Data Space}
\end{align*}
\]

Description:

In compatibility mode, this PDP-11 instruction works like a POP instruction. The destination operand is replaced by a word popped from the stack.

Note:

The implied source operand specifier is evaluated before the destination specifier.
MFP     Move From Previous Space

Format:

\[
\begin{array}{cccc}
1 & 5 & 5 & 0 \\
\hline
| \text{Opcode} & | \text{src.rw} | \\
\end{array}
\]

Operation:

\[-(\text{SP}) \leftarrow \text{src};\]

Condition Codes:

\[-N \leftarrow \text{src LSS 0};\]
\[-Z \leftarrow \text{src EQL 0};\]
\[-V \leftarrow 0;\]
\[-C \leftarrow C;\]

Exceptions:

none

Opcodes (octal):

\[
\begin{array}{lll}
0065 & \text{MFPI} & \text{Move From Previous Instruction Space} \\
1065 & \text{MPPD} & \text{Move From Previous Data Space} \\
\end{array}
\]

Description:

In compatibility mode, this PDP-11 instruction works like a \textsc{push} instruction. The source operand is pushed onto the stack.
CC     Condition Code Operators

Format:

```
       1
      5 4 0
+--------------------------+
| Opcode   |   mask   |
+--------------------------+
```

Operation:

```
if mask<4> EQL 1 then PSW<3:0> <- PSW<3:0> OR mask<3:0>
else PSW<3:0> <- PSW<3:0> AND {NOT mask<3:0>};
```

Condition Codes:

```
if mask<4> EQL 1 then
   begin
      N <- N OR mask<3>;
      Z <- Z OR mask<2>;
      V <- V OR mask<1>;
      C <- C OR mask<0>;
   end
else
   begin
      N <- N AND {NOT mask<3>};
      Z <- Z AND {NOT mask<2>};
      V <- V AND {NOT mask<1>};
      C <- C AND {NOT mask<0>};
   end
```

Exceptions:

none

Opcodes (octal):

```
000240  No operation
000241  CLC  Clear C
000242  CLV  Clear V
000244  CLZ  Clear Z
000250  CLN  Clear N
000257  CCC  Clear all Condition Codes
000261  SEC  Set C
000262  SEV  Set V
000264  SEZ  Set Z
000270  SEN  Set N
000277  SCC  Set all Condition Codes
```

Combinations of the above set or clear operations may be
ORed together to form combined instructions.

Description:

If the mask<4> bit is set, the PSW condition code bits are ORed with mask<3:0> and the condition codes are replaced by the result. If the mask<4> bit is clear, the PSW condition code bits are ANDed with the 1's complement of mask<3:0> and the condition codes are replaced by the result.
10.3 ENTERING AND LEAVING COMPATIBILITY MODE

Compatibility mode is entered by executing an REI instruction with the compatibility mode bit set in the image of the PSL on the stack. Other bits in the PSL have the following effects:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>NZVC</td>
<td>Condition Codes</td>
</tr>
<tr>
<td>T</td>
<td>T Bit</td>
</tr>
<tr>
<td>DV</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>FU</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>IV</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>IPL</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>PRV MOD</td>
<td>Reserved operand fault if not 3</td>
</tr>
<tr>
<td>CUR MOD</td>
<td>Reserved operand fault if not 3</td>
</tr>
<tr>
<td>IS</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>FPD</td>
<td>Reserved operand fault if not zero</td>
</tr>
<tr>
<td>TP</td>
<td>T pending bit. See Section on T bit operation in compatibility mode for a complete description of how trace faults work in compatibility mode.</td>
</tr>
</tbody>
</table>

VAX native mode is re-entered from compatibility mode by the compatibility mode program causing an exception, or by an interrupt. The PSL pushed on the kernel or interrupt stack when leaving compatibility mode has all the bits that cause reserved operand faults in the above table set to the appropriate state.

Note that when an RTI or RTT instruction is executed in compatibility mode, the 11 high bits of the PSW are ignored, but when the PSW is restored as part of the PSL when going from VAX native mode to compatibility mode, those bits must be zero, or a reserved operand fault occurs.

10.3.1 General Register Usage

Compatibility mode registers R0 through R6 are bits 15 through 0 of VAX general registers R0 through R6, respectively. Compatibility mode
register R7 (PC) is bits 15 through 0 of VAX general register R15 (PC). VAX registers R8 through R14 (SP) are not affected by compatibility mode. When entering compatibility mode, VAX register R7 and the upper halves of registers R0 through R6 and R15 are ignored. When an exception or interrupt occurs from compatibility mode, VAX register R7 is UNPREDICTABLE and the upper halves of R0 through R6 are either cleared or left unchanged. and the upper half of the stacked R15 (PC) is zero. Since there are no FP11 floating point instructions in compatibility mode, there are no floating accumulators.
10.4 COMPATIBILITY MODE MEMORY MANAGEMENT

PDP-11 addresses are 15-bit byte addresses, hence compatibility mode programs are confined to execute in the first 64k bytes of the per process part of the virtual address space. There is a one-to-one correspondence between a compatibility mode virtual address and its VAX counterpart (e.g., virtual address 0 references the same location in both modes). A compatibility mode address is interpreted as follows:

```
| 31 | 16 15 | 9 8 | 0 |
```

PDP-11 segments can consist of 1 to 128 blocks of 64 bytes. VAX pages are 512 bytes long. The PDP-11 capability of providing different access protection to different segments is provided in 8 block chunks since protection is specified at the page level in the VAX architecture.

The memory management system protects and relocates compatibility mode addresses in the normal native mode manner. Thus, all of the memory management mechanisms available in VAX mode are available to the compatibility mode executive for managing both the virtual and physical memory of compatibility mode programs. All of the exception conditions that can be caused by memory management in VAX mode can also occur when relocating a compatibility mode address. See Chapter 5.

Most of the KT-11 features that affect the user environment can be simulated with the VAX memory management system. Table 10-4 briefly describes the simulation method. Refer to Chapter 5 of this manual and the appropriate PDP-11 documents for details of each system.
Table 10-4

<table>
<thead>
<tr>
<th>KT11-D</th>
<th>VAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature to be simulated</td>
<td>simulation method</td>
</tr>
<tr>
<td>8 segments per user.</td>
<td>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.</td>
</tr>
<tr>
<td>Segment size from 54 to 9K bytes (1 to 128 blocks) in 54 byte increments, using contiguous memory.</td>
<td>Segment size from 512 to 8K bytes (1 to 16 pages) in 512 byte (1 page) increments, using discontiguous memory.</td>
</tr>
<tr>
<td>Forward growing segments (Expand Direction=0).</td>
<td>Can be simulated using page table entries specifying no access for those pages that are not allocated.</td>
</tr>
<tr>
<td>Backward growing segments (ED=1).</td>
<td>Can be simulated using page table entries specifying no access for those pages that are not allocated.</td>
</tr>
<tr>
<td>Segments begin on any 64 byte boundary.</td>
<td>Segments begin on any 512 byte boundary.</td>
</tr>
</tbody>
</table>
The following example shows how a PDP-11 environment can be simulated using VAX memory management. Segments 0, 1, and 2 of the PDP-11 environment are program segments; 3 is unused; 4 and 5 are stack; and 6 and 7 are read-write data.

### 11 Environment

<table>
<thead>
<tr>
<th>Seg #</th>
<th>Size (bytes)</th>
<th>Expand Direction</th>
<th>Access</th>
<th>Page</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8K</td>
<td>Up</td>
<td>Read only</td>
<td>0-15</td>
<td>Read only</td>
</tr>
<tr>
<td>1</td>
<td>8K</td>
<td>Up</td>
<td>Read only</td>
<td>16-31</td>
<td>Read only</td>
</tr>
<tr>
<td>2</td>
<td>255</td>
<td>Up</td>
<td>Read only</td>
<td>32</td>
<td>Read only</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>--</td>
<td>None</td>
<td>33-77</td>
<td>No Access</td>
</tr>
<tr>
<td>4</td>
<td>1K</td>
<td>Down</td>
<td>Read-Write</td>
<td>78-79</td>
<td>Read-Write</td>
</tr>
<tr>
<td>5</td>
<td>2K</td>
<td>Down</td>
<td>Read-Write</td>
<td>80-95</td>
<td>Read-Write</td>
</tr>
<tr>
<td>6</td>
<td>8K</td>
<td>Up</td>
<td>Read-Write</td>
<td>96-111</td>
<td>Read-Write</td>
</tr>
<tr>
<td>7</td>
<td>2K</td>
<td>Up</td>
<td>Read-Write</td>
<td>112-115</td>
<td>Read-Write</td>
</tr>
</tbody>
</table>

### VAX Page Table

<table>
<thead>
<tr>
<th>Page</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15</td>
<td>Read only</td>
</tr>
<tr>
<td>16-31</td>
<td>Read only</td>
</tr>
<tr>
<td>32</td>
<td>Read only</td>
</tr>
<tr>
<td>33-77</td>
<td>No Access</td>
</tr>
<tr>
<td>78-79</td>
<td>Read-Write</td>
</tr>
<tr>
<td>80-95</td>
<td>Read-Write</td>
</tr>
<tr>
<td>96-111</td>
<td>Read-Write</td>
</tr>
<tr>
<td>112-115</td>
<td>Read-Write</td>
</tr>
<tr>
<td>116-127</td>
<td>No Access</td>
</tr>
</tbody>
</table>
10.5 COMPATIBILITY MODE EXCEPTIONS AND INTERRUPTS

All interrupts and exception conditions that occur while the machine is in compatibility mode cause the machine to enter VAX mode, and are serviced as indicated in Chapter 6 (note that this includes backing up instruction side effects if necessary). The exception conditions discussed in this section are specific to compatibility mode. All these exceptions create a 3-longword frame on the kernel stack containing PSL, PC, and one longword of exception specific information. Bits 15 through 0 of this longword contain a code indicating the specific type of exception and bits 31 through 16 are zero. There are no compatibility mode exception conditions that result in traps (see Chapter 6 for definition of trap, fault, and abort).

10.5.1 Reserved Instruction Fault

A reserved instruction fault occurs for opcodes that are defined as reserved in compatibility mode (see section on Instructions). The code for the reserved instruction fault is 0.

10.5.2 BPT Instruction Fault

The code for the BPT instruction fault is 1.

10.5.3 IOT Instruction Fault

The code for the IOT instruction fault is 2.

10.5.4 EMT Instruction Fault

The fault code for the group of EMT instructions is 3.

10.5.5 TRAP Instruction Fault

The fault code for the group of TRAP instructions is 4.

10.5.6 Illegal Instruction Fault

In compatibility mode, JMP and JSR instructions with a register destination are illegal. The fault code for illegal instructions is 5.
10.5.7 Odd Address Error Abort

An odd address error abort is caused in compatibility mode whenever a word reference is attempted on a byte boundary. The code for odd address errors is 5.

10.6 **T BIT OPERATION IN COMPATIBILITY MODE**

In compatibility mode, a trace fault occurs at the beginning of an instruction when the T bit is set in the PSW at the beginning of the prior instruction. This effect is achieved by using the TP bit in the PSL (see Chapter 5). On trace faults, a 2-longword kernel stack frame is created, containing PSL and PC. IPL and IS are zero and CM is one in the stacked PSL. Compatibility mode trace fault uses the same vector as VAX mode trace fault. See Chapter 5. The rules for trace fault generation in compatibility mode are identical to those for native mode. However, an odd address abort for an instruction fetch may precede the trace fault for that instruction.

There are two ways to get the T bit set at the beginning of a compatibility mode instruction:

1. An RTT/RTI instruction is executed in compatibility mode with the T bit set in the PSW image on the stack. In this case, the next instruction is executed (the one pointed to by the PC on the stack), and a trace fault is taken before the following instruction.

2. An REI instruction is executed in VAX mode which has both the T bit and CM bit set (and TP clear) in the saved PSL image on the stack. Again, one instruction is executed, and the trace fault is taken. (For a complete description of the interaction of REI, T bit, and TP bit, see Chapter 6. The operations that occur as a function of these conditions are the same whether or not compatibility mode is being entered from the REI.)
The T bit interacts with other compatibility mode operations as follows (for interaction with other than compatibility mode specific operations, see Chapter 6):

1. T bit set (but TP is clear) at the beginning of any compatibility mode instruction which does not cause a compatibility mode fault.

In this case, the instruction sets TP and executes. A trace fault is taken before the next instruction. The saved PSL has the T bit set and TP clear. The compatibility mode executive can take one of the following courses of action:

1. If it services the exception directly, it can clear the T bit in the saved PSL on the kernel stack if it no longer wants to trace the program, or it can leave it set if it wants to continue tracing the program. It exits with an REI.

2. If it returns the trace exception to compatibility mode, it pushes a (16-bit) PC and (16-bit) PSW with the T bit set on the compatibility mode User stack to simulate the effect of the PDP-11 trace trap. It then clears the T bit in the saved PSL image on the kernel stack, changes the saved PC to point to the compatibility mode service routine, and does an REI. The compatibility mode service routine can then clear the T bit in the PSW image on its stack if it does not want to continue tracing. The compatibility mode routine returns with RTT or RTI.

2. T bit set (but TP is clear) at the beginning of an RTI or RTT.

The RTT/RTI instruction executes and TP is set. A trace fault occurs before the next instruction is executed. There are two different cases, depending on whether or not the T bit was set in the image of the PSW which was popped from the stack by the RTT/RTI instruction:

1. T bit not set.

Neither TP nor T will be set in the saved PSL on the kernel stack.

2. T bit set.

TP will not be set, and T will be set, as is the case as for other compatibility mode instructions.

3. T bit set (but TP is clear) at the beginning of any instruction which causes a compatibility mode fault.
The fault condition is serviced first. TP is clear and T is set in the saved PSL pushed on the kernel stack.

10.7 UNIMPLEMENTED PDP-11 TRAPS

Several traps that occur in PDP-11s are not implemented in compatibility mode:

1. There is no stack overflow trap. This is equivalent to the User Mode of the KTI1, where there is also no overflow protection. Stack overflow can be provided by the compatibility mode executive using the memory management mechanisms.

2. There is no concept of a double error trap in compatibility mode, since the first error always puts the machine in VAX mode.

3. All other exception conditions such as power failure, memory parity, and memory management exceptions cause the machine to enter VAX mode.
10.8 COMPATIBILITY MODE I/O REFERENCES

Neither instruction stream references nor data reads nor writes can be to I/O space. The results are UNPREDICTABLE if I/O space is referenced from compatibility mode.

10.9 PROCESSOR REGISTERS

The only processor register available in compatibility mode is part of the PSW, and it maybe explicitly referenced only with the condition code instructions, RTI, and RTT. Access to all other registers must be done in VAX mode.

10.10 PROGRAM SYNCHRONIZATION

All PDP-11s guarantee that read-modify-write operations to I/O device registers are interlocked; that is, the device can determine at the time of the read that the same register will be written as the next bus cycle. This synchronization also works in memory on most PDP-11s. In compatibility mode, instructions that have modify destinations will perform this synchronization for UNIBUS I/O device registers and never for memory.
APPENDIX A
INSTRUCTION SET AND OPCODE ASSIGNMENTS

23-Mar-81 -- Rev 17.1

A.1 INSTRUCTION OPERAND FORMATS

The format of the instructions is given using the qualified name convention described in the next section. For the mnemonics {} encloses a list of data types of which one must be selected. Instructions which have two forms differing in the number of operands have the number of operands appended to the opcode as a digit. For the operands, {} encloses all implied operands. Refer to the VAX-11 Macro Reference Manual for a description of when the data type suffix and operand number suffix may be omitted.

Instructions
----------

1. Move
   MOV{B,W,L,F,D,G,H,Q,O} src.rx, dst.wx
   9

2. Push Long
   PUSHL src.r1, {-(SP).w1}
   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
   40
   All pairs except BB,WW,LL,PF,DD,GG,HH,DG, and GD
8. Convert Rounded
   CVTR[F,D,G,H]L src.rx, dst.wl

9. Compare
   CMP[B,W,L,F,D,G,H] src1.rx, src2.rx

10. Test
    TST[B,W,L,F,D,G,H] src.rx

11. Add 2 Operand
    ADD[B,W,L,F,D,G,H]2 add.rx, sum.mx

12. Add 3 Operand
    ADD[B,W,L,F,D,G,H]3 add1.rx, add2.rx, sum.wx

13. Increment
    INC[B,W,L] sum.mx

14. Add With Carry
    ADWC add.rl, sum.ml

15. Add Aligned Word
    ADAWI add.rw, sum.mw

16. Subtract 2 Operand
    SUB[B,W,L,F,D,G,H]2 sub.rx, dif.mx

17. Subtract 3 Operand
    SUB[B,W,L,F,D,G,H]3 sub.rx, min.rx, dif.wx

18. Decrement
    DEC[B,W,L] dif.mx

19. Subtract With Carry
    SBWC sub.rl, dif.ml

20. Multiply 2 Operand
    MUL[B,W,L,F,D,G,H]2 mulr.rx, prod.mx

21. Multiply 3 Operand
    MUL[B,W,L,F,D,G,H]3 mulr.rx, muld.rx, prod.wx

22. Extended Multiply
    EMUL mulr.rl, muld.rl, add.rl, prod.wq

23. Divide 2 Operand
    DIV[B,W,L,F,D,G,H]2 divr.rx, quo.mx

24. Divide 3 Operand
    DIV[B,W,L,F,D,G,H]3 divr.rx, divd.rx, quo.wx

25. Extended Divide
    EDIV divr.rl, divd.rq, quo.wl, rem.wl
26. Arithmetic Shift
   ASH[L,Q] cnt.rb, src.rx, dst.wx
27. Bit Test
   BIT[B,W,L] mask.rx, src.rx
28. Bit Set 2 Operand
   BIS[B,W,L]2 mask.rx, dst.mx
29. Bit Set 3 Operand
   BIS[B,W,L]3 mask.rx, src.rx, dst.wx
30. Bit Clear 2 Operand
   BIC[B,W,L]2 mask.rx, dst.mx
31. Bit Clear 3 Operand
   BIC[B,W,L]3 mask.rx, src.rx, dst.wx
32. Exclusive OR 2 Operand
   XOR[B,W,L]2 mask.rx, dst.mx
33. Exclusive OR 3 Operand
   XOR[B,W,L]3 mask.rx, src.rx, dst.wx
34. Rotate Long
   ROTL cnt.rb, src.rl, dst.wl
35. Extended Modulus
   EMOD[F,D] mulr.rx, mulrx.rb, muld.rx, int.w1, fract.wx
   EMOD[G,H] mulr.rx, mulrx rw, muld.rx, int.w1, fract.wx
36. Polynomial Evaluation F_floating
   POLYF arg.rf, degree.rw, tbladdr.ab, [R0-3.w1]
37. Polynomial Evaluation D_floating
   POLYD arg.rd, degree.rw, tbladdr.ab, [R0-5.w1]
38. Polynomial Evaluation G_floating
   POLYG arg.rg, degree.rw, tbladdr.ab, [R0-5.w1]
39. Polynomial Evaluation H_floating
   POLYH arg.rh, degree.rw, tbladdr.ab,
   {R0-5.w1,-16(SP):-1(SP).wb}
40. Move Address
   MOVA[B,W,L=F,Q=D=G,O=H] src.ax, dst.wl
41. Push Address
42. Index
   INDEX subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl
43. Extract Field
   EXTV pos.rl, size.rb, base.vb, {field.rv}, dst.wl

44. Extract Zero-Extended Field
   EXTZV pos.rl, size.rb, base.vb, {field.rv}, dst.wl

45. Insert Field
   INSV src.rl, pos.rl, size.rb, base.vb, {field.wv}

46. Compare Field
   CMPV pos.rl, size.rb, base.vb, {field.rv}, src.rl

47. Compare Zero-Extended Field
   CMP2V pos.rl, size.rb, base.vb, {field.rv}, src.rl

48. Find First
   FF{S,C} startpos.rl, size.rb, base.vb, {field.rv}, findpos.wl

49. Conditional Branch
   B{condition} displ.bb

<table>
<thead>
<tr>
<th>Condition</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSS</td>
<td>Less Than</td>
</tr>
<tr>
<td>LEQ</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>EQL, EQLU</td>
<td>Equal, Equal Unsigned</td>
</tr>
<tr>
<td>NEQ, NEQU</td>
<td>Not Equal, Not Equal Unsigned</td>
</tr>
<tr>
<td>GEQ</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>GTR</td>
<td>Greater Than</td>
</tr>
<tr>
<td>LSSU, CS</td>
<td>Less Than Unsigned, Carry Set</td>
</tr>
<tr>
<td>LEQU</td>
<td>Less Than or Equal Unsigned</td>
</tr>
<tr>
<td>GEQU, CC</td>
<td>Greater Than or Equal Unsigned, Carry Clear</td>
</tr>
<tr>
<td>GTRU</td>
<td>Greater Than Unsigned</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow Set</td>
</tr>
<tr>
<td>VC</td>
<td>Overflow Clear</td>
</tr>
</tbody>
</table>

50. Branch With {Byte, Word} Displacement
    BR{B,W} displ.bx

51. Jump
    JMP dst.ab

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

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

54. Branch on Bit (and modify) Interlocked
    BB{SS,CC}I pos.rl, base.vb, displ.bb, {field.mv}
55. Branch on Low Bit
   BLB[S,C] src.rl, displ.bb

56. Add Compare and Branch
   ACB[B,W,L,F,D,G,H] limit.rx, add.rx, index.mx, displ.bw
   Compare is LE on positive add, GE on negative add.

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

58. Add One and Branch Less Than
   AOBQSS limit.rl, index.ml, displ.bb

59. Subtract One and Branch Greater Than or Equal
   SOBGEQ index.ml, displ.bb

60. Subtract One and Branch Greater Than
   SOBGTR index.ml, displ.bb

61. Case
   CASE[B,W,L] selector.rx, base.rx, limit.rx, displ.bw-list

62. Branch to Subroutine With {Byte, Word} Displacement
   BSB[B,W] displ.bx, {-(SP).wl}

63. Jump to Subroutine
   JSB dst.ab, {-(SP).wl}

64. Return from Subroutine
   RSB {(SP)+.rl}

65. Call Procedure with General Argument List
   CALLG arglist.ab, dst.ab, {-(SP).w*}

66. Call Procedure with Stack Argument List
   CALLS numarg.rl, dst.ab, {-(SP).w*}

67. Return from Procedure
   RET {(SP)+.r*}

68. Breakpoint Fault
   BPT {-(KSP).w*}

69. Halt
   HALT {-(KSP).w*}
   Halts in Kernel mode, faults otherwise.
   Assigned opcode 0.

70. Push Registers
   PUSHR mask.rw, {-(SP).w*}
71. Pop Registers
   POPR mask.rw, {(SP)+.r*}  1

72. Move from PSL
   MOVPSL dst.w1  1

73. Bit Set PSW
   BISPSW mask.rw  1

74. Bit Clear PSW
   BICPSW mask.rw  1

75. No Operation
   NOP  1

76. Extended Function Call
   XFC {unspecified operands}  1

77. Insert Entry in Queue
   INSQUE entry.ab, pred.ab  1

78. Insert Entry into Queue at Head, Interlocked
   INSQHI entry.ab, header.aq  1

79. Insert Entry into Queue at Tail, Interlocked
   INSQTI entry.ab, header.aq  1

80. Remove Entry from Queue
   REMQUE entry.ab, addr.w1  1

81. Remove Entry from Queue at Head, Interlocked
   REMQHI header.aq, addr.w1  1

82. Remove Entry from Queue at Tail, Interlocked
   REMQTI header.aq, addr.w1  1

83. Move Character 3 Operand
   MOVCC2 len.rw, srcaddr.ab, dstaddr.ab, [R0-5.wl]  1

84. Move Character 5 operand
   MOVCC5 srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab, [R0-5.wl]  1

85. Move Translated Characters
   MOVTC srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab, [R0-5.wl]  1

86. Move Translated Until Character
   MOVTUC srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw, dstaddr.ab, [R0-5.wl]  1

87. Compare Characters 3 Operand
   CMPC3 len.rw, src1addr.ab, src2addr.ab, [R0-3.wl]  1
88. Compare Characters 5 Operand
   CMPC5 src1llen.rw, src1laddr.ab, fill.rb, src2llen.rw,
   src2addr.ab, [R0-3.w1]

89. Scan Characters
   SCANC len.rw, addr.ab, tbladdr.ab, mask.rb, [R0-3.w1]

90. Span Characters
   SPANC len.rw, addr.ab, tbladdr.ab, mask.rb, [R0-3.w1]

91. Locate Character
   LOCC char.rb, len.rw, addr.ab, [R0-1.w1]

92. Skip Character
   SKPC char.rb, len.rw, addr.ab, [R0-1.w1]

93. Match Characters
   MATCHC len1.rw, addr1.ab, len2.rw, addr2.ab, [R0-3.w1]

94. Cyclic Redundancy Check
   CRC tbl1.ab, iniirc.r1, strlen.rw, stream.ab, [R0-3.w1]

95. Move Packed
   MOV P len.rw, srcaddr.ab, dstaddr.ab, [R0-3.w1]

96. Compare Packed 3 Operand
   CMPP3 len.rw, srcladdr.ab, src2addr.ab, [R0-3.w1]

97. Compare Packed 4 Operand
   CMPP4 src1llen.rw, src1laddr.ab, src2llen.rw, src2addr.ab,
   [R0-3.w1]

98. Add Packed 4 Operand
   ADDP4 addllen.rw, addaddr.ab, sumllen.rw, sumaddr.ab, [R0-3.w1]

99. Add Packed 6 Operand
   ADDP6 add1llen.rw, add1addr.ab, add2llen.rw, add2addr.ab,
   sumllen.rw, sumaddr.ab, [R0-5.w1]

100. Subtract Packed 4 Operand
    SUBP4 subllen.rw, subaddr.ab, difllen.rw, difaddr.ab, [R0-3.w1]

101. Subtract Packed 6 Operand
    SUBP6 subllen.rw, subaddr.ab, minllen.rw, minaddr.ab,
    difllen.rw, difaddr.ab, [R0-5.w1]

102. Multiply Packed
    MULP mulllen.rw, mulraddr.ab, muldllen.rw, muldaddr.ab,
    prodllen.rw, prodaddr.ab, [R0-5.w1]

103. Divide Packed
    DIVP divrlen.rw, divraddr.ab, divdllen.rw, divdaddr.ab,
    quolen.rw, quoadaddr.ab, [R0-5.w1, -16(SP):-1(SP).wb]
104. Convert Long to Packed
    CVTLP src.rl, dstlen.rw, dstaddr.ab, {R0-3.wl}

105. Convert Packed to Long
    CVTPL srclen.rw, srcaddr.ab, {R0-3.wl}, dst.wl

106. Convert Packed to Trailing
    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
    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
    ASHP cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw,
    dstaddr.ab, {R0-3.wl}

109. Edit Packed to Character String
    EDITPC srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab, {R0-5.wl}

110. Probe {Read, Write} Accessability
    PROBE{R,W} mode.rb, len.rw, base.ab

111. Change Mode
    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
    REI {(SP)+.r*}  

113. Load Process Context
    LDPCTX {PCB.r*, -(KSP).w*}  
    Legal only on interrupt stack.

114. Save Process Context
    SVPCTX {(SP)+.r*, PCB.w*}  
    Legal only in Kernel mode.

115. Move To Process Register
    MTPR src.rl, procreg.rl  
    Legal only in Kernel mode.

116. Move From Processor Register
    MFPR procreg.rl, dst.wl  
    Legal only in Kernel mode.

Total 304
A.2 OPERAND SPECIFIER NOTATION

The standard VAX notation for operand specifiers is:

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

where:

1. Name is a suggestive name for the operand in the context of the instruction. It is the capitalized name of a register or block for implied operands.

2. Access type is a letter denoting the operand specifier access type.
   a - Calculate the effective address of the specified operand. Address is returned in a pointer which is the actual instruction operand. Context of address calculation is given by data type given by <data type>.
   b - No operand reference. Operand specifier is branch displacement. Size of branch displacement is given by <data type>.
   m - operand is modified (both read and written)
   r - operand is read only
   v - if not "Rn", same as a. If "Rn", R[n+1]'R[n].
   w - operand is written only

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

   b - byte
   d - D_floating
   f - F_floating
   g - G_floating
   h - H_floating
   l - longword
   o - octaword
   q - quadword
   v - field (used only on implied operands)
   w - word
   x - first data type specified by instruction
   y - second data type specified by instruction
   * - multiple longwords (used only on implied operands)

For names, the following names and abbreviations are used:

1. add - addend
2. addr - address
3. arglist - argument list
4. base - base
5. char - character
6. cnt - count
7. dif - difference
8. displ - displacement
9. divd - dividend
10. divr - divisor
11. dst - destination
12. entry - entry
13. esc - escape
14. fill - fill
15. findpos - find position
16. fract - fraction
17. index - index
18. inicrc - initial crc
19. int - integer
20. len - length
21. limit - limit
22. mask - mask
23. min - minuend
24. muld - multiplicand
25. mulr - multiplier
26. mulrx - multiplier extension
27. numarg - number of arguments
28. option - option
29. param - parameter
30. pos - position
31. pred - predecessor
32. procreg - internal processor register
33. prod - product
34. quo - quotient
35. rem - remainder
36. selector - selector
37. size - size
38. src - source
39. startpos - starting position
40. stream - stream
41. strlen - string length
42. sub - subtrahend
43. sum - sum
44. tbl - table
### Single Byte Opcodes

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Mnemonic</th>
<th>Binary</th>
<th>Hex</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00</td>
<td>HALT</td>
<td>00100000</td>
<td>20</td>
<td>ADDP4</td>
</tr>
<tr>
<td>00000001</td>
<td>01</td>
<td>NOP</td>
<td>00100001</td>
<td>21</td>
<td>ADDP6</td>
</tr>
<tr>
<td>00000010</td>
<td>02</td>
<td>REI</td>
<td>00100100</td>
<td>22</td>
<td>SUBP4</td>
</tr>
<tr>
<td>00000111</td>
<td>03</td>
<td>BPT</td>
<td>00100111</td>
<td>23</td>
<td>SUBP6</td>
</tr>
<tr>
<td>00001000</td>
<td>04</td>
<td>RET</td>
<td>00101000</td>
<td>24</td>
<td>CVTPT</td>
</tr>
<tr>
<td>00001010</td>
<td>05</td>
<td>RSB</td>
<td>00101010</td>
<td>25</td>
<td>MULP</td>
</tr>
<tr>
<td>00001100</td>
<td>06</td>
<td>LDCTX</td>
<td>00101011</td>
<td>26</td>
<td>CVTTP</td>
</tr>
<tr>
<td>00001111</td>
<td>07</td>
<td>SVPCTX</td>
<td>00101111</td>
<td>27</td>
<td>DIVP</td>
</tr>
<tr>
<td>00010000</td>
<td>08</td>
<td>CVTPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010001</td>
<td>09</td>
<td>CVTSP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010100</td>
<td>0A</td>
<td>INDEX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010110</td>
<td>0B</td>
<td>CRC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011010</td>
<td>0C</td>
<td>PROBER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011011</td>
<td>0D</td>
<td>PROBEW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011110</td>
<td>0E</td>
<td>INSQWE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011111</td>
<td>0F</td>
<td>REMQUE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100000</td>
<td>10</td>
<td>BSBB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100001</td>
<td>11</td>
<td>BRB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100100</td>
<td>12</td>
<td>BNEQ,BNEQU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100110</td>
<td>13</td>
<td>BEQL,BEQU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101000</td>
<td>14</td>
<td>BGTR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101010</td>
<td>15</td>
<td>BLEQ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101110</td>
<td>16</td>
<td>JSB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110111</td>
<td>17</td>
<td>JMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110000</td>
<td>18</td>
<td>BGEQ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110010</td>
<td>19</td>
<td>BLSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110100</td>
<td>1A</td>
<td>BGTRU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111011</td>
<td>1B</td>
<td>BLEQU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111000</td>
<td>1C</td>
<td>BVC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111101</td>
<td>1D</td>
<td>BVS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111100</td>
<td>1E</td>
<td>BGEQU,BCC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111111</td>
<td>1F</td>
<td>BLSSU,BCS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Set and Opcode Assignments

**OPCODE ASSIGNMENTS**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Mnemonic</th>
<th>Binary</th>
<th>Hex</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000000 40</td>
<td>ADDF2</td>
<td>01100000 60</td>
<td>ADDD2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000001 41</td>
<td>ADDF3</td>
<td>01100001 61</td>
<td>ADDD3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000010 42</td>
<td>SUBF2</td>
<td>01100010 62</td>
<td>SUBD2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000011 43</td>
<td>SUBF3</td>
<td>01100011 63</td>
<td>SUBD3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000100 44</td>
<td>MULF2</td>
<td>01100100 64</td>
<td>MULD2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000101 45</td>
<td>MULF3</td>
<td>01100101 65</td>
<td>MULD3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000110 46</td>
<td>DIVF2</td>
<td>01100110 66</td>
<td>DIVD2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000111 47</td>
<td>DIVF3</td>
<td>01100111 67</td>
<td>DIVD3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001000 48</td>
<td>CVTFB</td>
<td>01101000 68</td>
<td>CVTBD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001001 49</td>
<td>CVTFW</td>
<td>01101001 69</td>
<td>CVTDW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001010 4A</td>
<td>CVTFL</td>
<td>01101010 6A</td>
<td>CVTDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001011 4B</td>
<td>CVTRFL</td>
<td>01101011 6B</td>
<td>CVTRDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001100 4C</td>
<td>CVTFB</td>
<td>01101100 6C</td>
<td>CVTBD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001101 4D</td>
<td>CVTWF</td>
<td>01101101 6D</td>
<td>CVTWD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001110 4E</td>
<td>CVTFL</td>
<td>01101110 6E</td>
<td>CVTLD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001111 4F</td>
<td>ACBF</td>
<td>01101111 6F</td>
<td>ACBD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010000 50</td>
<td>MOVF</td>
<td>01110000 70</td>
<td>MOVD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010001 51</td>
<td>CMPF</td>
<td>01110001 71</td>
<td>CMPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010010 52</td>
<td>MNEGF</td>
<td>01110010 72</td>
<td>MNEGD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010011 53</td>
<td>TSTF</td>
<td>01110011 73</td>
<td>TSTD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010100 54</td>
<td>EMODF</td>
<td>01110100 74</td>
<td>EMODD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010101 55</td>
<td>POLYF</td>
<td>01110101 75</td>
<td>POLYD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010110 56</td>
<td>CVTFD</td>
<td>01110110 76</td>
<td>CVTDF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010111 57</td>
<td>RESERVED to DEC</td>
<td>01110111 77</td>
<td>RESERVED to DEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011000 58</td>
<td>ADAWI</td>
<td>01111000 78</td>
<td>ASHL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011001 59</td>
<td>RESERVED to DEC</td>
<td>01111001 79</td>
<td>ASHQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011010 5A</td>
<td>RESERVED to DEC</td>
<td>01111010 7A</td>
<td>EMUL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011011 5B</td>
<td>RESERVED to DEC</td>
<td>01111011 7B</td>
<td>EDIV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011100 5C</td>
<td>INSqHI</td>
<td>01111100 7C</td>
<td>CLRQ,CLRD,CLRG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011101 5D</td>
<td>INSqTI</td>
<td>01111101 7D</td>
<td>MOVQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011110 5E</td>
<td>REMQHI</td>
<td>01111110 7E</td>
<td>MOVAQ,MOVAD,MOVAG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011111 5F</td>
<td>REMQTI</td>
<td>01111111 7F</td>
<td>PUSAQ,PUSHAD,PUSHAG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>Hex</td>
<td>Mnemonic</td>
<td>Binary</td>
<td>Hex</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>----------</td>
<td>---------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>10000000</td>
<td>80</td>
<td>ADDB2</td>
<td>10100000</td>
<td>A0</td>
<td>ADDW2</td>
</tr>
<tr>
<td>10000001</td>
<td>81</td>
<td>ADDB3</td>
<td>10100010</td>
<td>A1</td>
<td>ADDW3</td>
</tr>
<tr>
<td>10000010</td>
<td>82</td>
<td>SUBB2</td>
<td>10100011</td>
<td>A2</td>
<td>SUBW2</td>
</tr>
<tr>
<td>10000011</td>
<td>83</td>
<td>SUBB3</td>
<td>10100100</td>
<td>A3</td>
<td>SUBW3</td>
</tr>
<tr>
<td>10000100</td>
<td>84</td>
<td>MULB2</td>
<td>10100101</td>
<td>A4</td>
<td>MULW2</td>
</tr>
<tr>
<td>10000101</td>
<td>85</td>
<td>MULB3</td>
<td>10101001</td>
<td>A5</td>
<td>MULW3</td>
</tr>
<tr>
<td>10000110</td>
<td>86</td>
<td>DIVB2</td>
<td>10101010</td>
<td>A6</td>
<td>DIVW2</td>
</tr>
<tr>
<td>10000111</td>
<td>87</td>
<td>DIVB3</td>
<td>10101011</td>
<td>A7</td>
<td>DIVW3</td>
</tr>
<tr>
<td>10001000</td>
<td>88</td>
<td>BISB2</td>
<td>10101000</td>
<td>A8</td>
<td>BISW2</td>
</tr>
<tr>
<td>10001001</td>
<td>89</td>
<td>BISB3</td>
<td>10101001</td>
<td>A9</td>
<td>BISW3</td>
</tr>
<tr>
<td>10001010</td>
<td>8A</td>
<td>BICB2</td>
<td>10101010</td>
<td>AA</td>
<td>BICW2</td>
</tr>
<tr>
<td>10001011</td>
<td>8B</td>
<td>BICB3</td>
<td>10101011</td>
<td>AB</td>
<td>BICW3</td>
</tr>
<tr>
<td>10001100</td>
<td>8C</td>
<td>XORB2</td>
<td>10101100</td>
<td>AC</td>
<td>XORW2</td>
</tr>
<tr>
<td>10001101</td>
<td>8D</td>
<td>XORB3</td>
<td>10101101</td>
<td>AD</td>
<td>XORW3</td>
</tr>
<tr>
<td>10001110</td>
<td>8E</td>
<td>MNEGB</td>
<td>10101110</td>
<td>AE</td>
<td>MNEGW</td>
</tr>
<tr>
<td>10001111</td>
<td>8F</td>
<td>CASEB</td>
<td>10101111</td>
<td>AF</td>
<td>CASEW</td>
</tr>
<tr>
<td>10010000</td>
<td>90</td>
<td>MOVB</td>
<td>10110000</td>
<td>B0</td>
<td>MOVW</td>
</tr>
<tr>
<td>10010001</td>
<td>91</td>
<td>CMPB</td>
<td>10110001</td>
<td>B1</td>
<td>CMPW</td>
</tr>
<tr>
<td>10010010</td>
<td>92</td>
<td>MCBMB</td>
<td>10110010</td>
<td>B2</td>
<td>MCBMW</td>
</tr>
<tr>
<td>10010011</td>
<td>93</td>
<td>BITB</td>
<td>10110011</td>
<td>B3</td>
<td>BITW</td>
</tr>
<tr>
<td>10010100</td>
<td>94</td>
<td>CLRBB</td>
<td>10110100</td>
<td>B4</td>
<td>CLRW</td>
</tr>
<tr>
<td>10010101</td>
<td>95</td>
<td>TSBB</td>
<td>10110101</td>
<td>B5</td>
<td>TSW</td>
</tr>
<tr>
<td>10010110</td>
<td>96</td>
<td>INCBB</td>
<td>10110110</td>
<td>B6</td>
<td>INCW</td>
</tr>
<tr>
<td>10010111</td>
<td>97</td>
<td>DECB</td>
<td>10110111</td>
<td>B7</td>
<td>DECW</td>
</tr>
<tr>
<td>10011000</td>
<td>98</td>
<td>CVTBL</td>
<td>10111000</td>
<td>B8</td>
<td>BISPW</td>
</tr>
<tr>
<td>10011001</td>
<td>99</td>
<td>CVTBW</td>
<td>10111001</td>
<td>B9</td>
<td>BICPSW</td>
</tr>
<tr>
<td>10011010</td>
<td>9A</td>
<td>MOVZBL</td>
<td>10111010</td>
<td>BA</td>
<td>POPR</td>
</tr>
<tr>
<td>10011011</td>
<td>9B</td>
<td>MOVZBW</td>
<td>10111011</td>
<td>BB</td>
<td>PUSR</td>
</tr>
<tr>
<td>10011100</td>
<td>9C</td>
<td>ROTL</td>
<td>10111100</td>
<td>BC</td>
<td>CHMK</td>
</tr>
<tr>
<td>10011101</td>
<td>9D</td>
<td>ACBB</td>
<td>10111101</td>
<td>BD</td>
<td>CHME</td>
</tr>
<tr>
<td>10011110</td>
<td>9E</td>
<td>MOVAB</td>
<td>10111110</td>
<td>BE</td>
<td>CHMS</td>
</tr>
<tr>
<td>10011111</td>
<td>9F</td>
<td>PUSHAB</td>
<td>10111111</td>
<td>BF</td>
<td>CHMU</td>
</tr>
</tbody>
</table>
### Instruction Set and Opcode Assignments

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Mnemonic</th>
<th>Binary</th>
<th>Hex</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000000</td>
<td>C0</td>
<td>ADDLL2</td>
<td>11100000</td>
<td>E0</td>
<td>BBS</td>
</tr>
<tr>
<td>11000001</td>
<td>C1</td>
<td>ADDL3</td>
<td>11100001</td>
<td>E1</td>
<td>BBC</td>
</tr>
<tr>
<td>11000010</td>
<td>C2</td>
<td>SUBLL2</td>
<td>11100010</td>
<td>E2</td>
<td>BBSS</td>
</tr>
<tr>
<td>11000011</td>
<td>C3</td>
<td>SUBL3</td>
<td>11100011</td>
<td>E3</td>
<td>BBCS</td>
</tr>
<tr>
<td>11000100</td>
<td>C4</td>
<td>MULL2</td>
<td>11100100</td>
<td>E4</td>
<td>BBSC</td>
</tr>
<tr>
<td>11000101</td>
<td>C5</td>
<td>MULL3</td>
<td>11100101</td>
<td>E5</td>
<td>BBCC</td>
</tr>
<tr>
<td>11000110</td>
<td>C6</td>
<td>DIVL2</td>
<td>11100110</td>
<td>E6</td>
<td>BBSSI</td>
</tr>
<tr>
<td>11000111</td>
<td>C7</td>
<td>DIVL3</td>
<td>11100111</td>
<td>E7</td>
<td>BBCCI</td>
</tr>
<tr>
<td>11001000</td>
<td>C8</td>
<td>BISL2</td>
<td>11101000</td>
<td>E8</td>
<td>BLBS</td>
</tr>
<tr>
<td>11001001</td>
<td>C9</td>
<td>BISL3</td>
<td>11101001</td>
<td>E9</td>
<td>BLBC</td>
</tr>
<tr>
<td>11001010</td>
<td>CA</td>
<td>BICL2</td>
<td>11101010</td>
<td>EA</td>
<td>FFS</td>
</tr>
<tr>
<td>11001011</td>
<td>CB</td>
<td>BICL3</td>
<td>11101011</td>
<td>EB</td>
<td>FPC</td>
</tr>
<tr>
<td>11001100</td>
<td>CC</td>
<td>XORL2</td>
<td>11101100</td>
<td>EC</td>
<td>CMPV</td>
</tr>
<tr>
<td>11001101</td>
<td>CD</td>
<td>XORL3</td>
<td>11101101</td>
<td>ED</td>
<td>CMPZV</td>
</tr>
<tr>
<td>11001110</td>
<td>CE</td>
<td>MNEGL</td>
<td>11101110</td>
<td>EE</td>
<td>EXTV</td>
</tr>
<tr>
<td>11001111</td>
<td>CF</td>
<td>CASEL</td>
<td>11101111</td>
<td>EF</td>
<td>EXTZV</td>
</tr>
<tr>
<td>11010000</td>
<td>D0</td>
<td>MOVL</td>
<td>11110000</td>
<td>F0</td>
<td>INSV</td>
</tr>
<tr>
<td>11010001</td>
<td>D1</td>
<td>CMPL</td>
<td>11110001</td>
<td>F1</td>
<td>ACBL</td>
</tr>
<tr>
<td>11010010</td>
<td>D2</td>
<td>MCOML</td>
<td>11110010</td>
<td>F2</td>
<td>ABLSS</td>
</tr>
<tr>
<td>11010011</td>
<td>D3</td>
<td>BTL</td>
<td>11110011</td>
<td>F3</td>
<td>ABOLEQ</td>
</tr>
<tr>
<td>11010100</td>
<td>D4</td>
<td>CLRLO,CLRF</td>
<td>11110100</td>
<td>F4</td>
<td>SOBGEQ</td>
</tr>
<tr>
<td>11010101</td>
<td>D5</td>
<td>TSTL</td>
<td>11110101</td>
<td>F5</td>
<td>SOBGR</td>
</tr>
<tr>
<td>11010110</td>
<td>D6</td>
<td>INCL</td>
<td>11110110</td>
<td>F6</td>
<td>CVTLB</td>
</tr>
<tr>
<td>11010111</td>
<td>D7</td>
<td>DECL</td>
<td>11110111</td>
<td>F7</td>
<td>CVTLW</td>
</tr>
<tr>
<td>11011000</td>
<td>D8</td>
<td>ADWC</td>
<td>11111000</td>
<td>F8</td>
<td>ASHP</td>
</tr>
<tr>
<td>11011001</td>
<td>D9</td>
<td>SBWC</td>
<td>11111001</td>
<td>F9</td>
<td>CVTLP</td>
</tr>
<tr>
<td>11011010</td>
<td>DA</td>
<td>MTPR</td>
<td>11111010</td>
<td>FA</td>
<td>CALLG</td>
</tr>
<tr>
<td>11011011</td>
<td>DB</td>
<td>MPPR</td>
<td>11111011</td>
<td>FB</td>
<td>CALLS</td>
</tr>
<tr>
<td>11011100</td>
<td>DC</td>
<td>MOVPSL</td>
<td>11111100</td>
<td>FC</td>
<td>XFC</td>
</tr>
<tr>
<td>11011101</td>
<td>DD</td>
<td>PUSHL</td>
<td>11111101</td>
<td>FD</td>
<td>ESCD to DEC</td>
</tr>
<tr>
<td>11011110</td>
<td>DE</td>
<td>MOVAL,MOVAF</td>
<td>11111110</td>
<td>FE</td>
<td>ESCE to DEC</td>
</tr>
<tr>
<td>11011111</td>
<td>DF</td>
<td>PUSHAL,PUSHAF</td>
<td>11111111</td>
<td>FF</td>
<td>ESCF to DEC</td>
</tr>
</tbody>
</table>
### Two Byte Opcodes

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Hex</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>00FD</td>
<td></td>
<td>33FD</td>
<td>CVTGPF</td>
</tr>
<tr>
<td>to</td>
<td>RESERVED</td>
<td></td>
<td>32FD</td>
</tr>
<tr>
<td>31FD</td>
<td>DIGITAL</td>
<td></td>
<td>33FD</td>
</tr>
<tr>
<td>32FD</td>
<td>CVTDH</td>
<td>34FD</td>
<td></td>
</tr>
<tr>
<td>34FD</td>
<td>RESERVED</td>
<td>34FD</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>to DEC</td>
<td>3FFD</td>
<td></td>
</tr>
<tr>
<td>32FD</td>
<td>CVTDH</td>
<td></td>
<td>33FD</td>
</tr>
<tr>
<td>34FD</td>
<td>RESERVED</td>
<td>34FD</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>to DEC</td>
<td>34FD</td>
<td></td>
</tr>
<tr>
<td>40FD</td>
<td>ADDG2</td>
<td>60FD</td>
<td>ADDH2</td>
</tr>
<tr>
<td>41FD</td>
<td>ADDG3</td>
<td>61FD</td>
<td>ADDH3</td>
</tr>
<tr>
<td>42FD</td>
<td>SUBG2</td>
<td>62FD</td>
<td>SUBH2</td>
</tr>
<tr>
<td>43FD</td>
<td>SUBG3</td>
<td>63FD</td>
<td>SUBH3</td>
</tr>
<tr>
<td>44FD</td>
<td>MULG2</td>
<td>64FD</td>
<td>MULH2</td>
</tr>
<tr>
<td>45FD</td>
<td>MULG3</td>
<td>65FD</td>
<td>MULH3</td>
</tr>
<tr>
<td>46FD</td>
<td>DIVG2</td>
<td>66FD</td>
<td>DIVH2</td>
</tr>
<tr>
<td>47FD</td>
<td>DIVG3</td>
<td>67FD</td>
<td>DIVH3</td>
</tr>
<tr>
<td>48FD</td>
<td>CVTGB</td>
<td>68FD</td>
<td>CVTHB</td>
</tr>
<tr>
<td>49FD</td>
<td>CVTGW</td>
<td>69FD</td>
<td>CVTHW</td>
</tr>
<tr>
<td>4AFD</td>
<td>CVTG1</td>
<td>6AFD</td>
<td>CVTHL</td>
</tr>
<tr>
<td>4BFD</td>
<td>CVTRGL</td>
<td>6BFD</td>
<td>CVTRHL</td>
</tr>
<tr>
<td>4CFD</td>
<td>CVTGB</td>
<td>6CFD</td>
<td>CVTBH</td>
</tr>
<tr>
<td>4DFD</td>
<td>CVTWG</td>
<td>6DFD</td>
<td>CVTWH</td>
</tr>
<tr>
<td>4EFD</td>
<td>CVTLEG</td>
<td>6EFD</td>
<td>CVTLH</td>
</tr>
<tr>
<td>4FFD</td>
<td>ACBG</td>
<td>6FFD</td>
<td>ACBH</td>
</tr>
</tbody>
</table>
50FD  MOVG
51FD  CMPG
52FD  MNENG
53FD  TSTG
54FD  EMODG
55FD  POLYG
56FD  CVTGH
57FD  RESERVED to DEC
58FD  RESERVED to DEC
59FD  RESERVED to DEC
5AFD  RESERVED to DEC
5BFD  RESERVED to DEC
5CFD  RESERVED to DEC
5DFD  RESERVED to DEC
5EFD  RESERVED to DEC
5FFD  RESERVED to DEC

70FD  MOVH
71FD  CMPH
72FD  MNEGH
73FD  TSTH
74FD  EMODH
75FD  POLYH
76FD  CVTHG
77FD  RESERVED to DEC
78FD  RESERVED to DEC
79FD  RESERVED to DEC
7AFD  RESERVED to DEC
7BFD  RESERVED to DEC
7CFD  CLRH,CLRO
7DFD  MOVO
7EFD  MOVAH,MOVAO
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

FFFFFF  RESERVED for all time
A.4 INSTRUCTIONS USABLE TO REFERENCE I/O SPACE

Some of the instructions are not usable to reference I/O space. The reasons for this are:

1. String instructions are restartable via PSL<FPD>
2. The instruction is not in the kernel set
3. The PC, SP, or PCBB can not point to I/O space
4. I/O space does not support operand types of quad, floating, field, or queue; nor can the position, size, length, or base of them be from I/O space
5. The instruction may be interruptible because it is potentially a slow instruction in some implementations
6. Only instructions with a maximum of one modify or write destination can be used. The destination must be the last operand

For any memory reference to I/O space, the programmer must use an instruction from the following lists and must ensure that no interrupts or faults will occur, including page faults, after the first I/O space reference. To ensure no interrupts, the programmer must avoid operand specifier modes 9, 11, 13, and 15, and these modes indexed. (Symbolically, these are @(Rn)+, @B<sub>D</sub>(Rn), @W<sub>D</sub>(Rn), and @L<sub>D</sub>(Rn), and these indexed.) The hardware may allow interrupts for these modes in order to minimize interrupt latency. For the instructions in the following lists, the hardware ensures that no other interrupts will occur after the first I/O space access.

Since these instructions are not interruptable after I/O space accesses (except for the addressing modes above), their execution will extend the interrupt latency. The programmer should make some effort to keep them short by minimizing the number of memory references. Use R0 through R13 instead, for example.

Instructions for which any explicit operand can be in I/O space:


Instructions for which all operands except the branch displacement can be in I/O space:

BLB[S,C]
Instruction for which some operand can be in I/O space:

XFC    (depending on implementation)
REMQUE  addr (destination)
REMQHI  addr (destination)
REMQTI  addr (destination)

Notwithstanding the above rules, it is possible for a specific hardware implementation to execute macro code from the I/O space and/or to allow the stack or PCB to be in I/O space. This might, for example, be used as part of the bootstrap process. If this is done, then it is valid for software to transfer to this code.
INDEX

() as a notation, 3-4
{} as a notation, 3-4

Abort, 6-1, 6-3
Absolute addressing mode, 3-7
Absolute indexed addressing mode, 3-14
Absolute indexed mode, 3-14
Absolute mode, 3-7
Interlocked,
Absolute queues, 4-90
ACBB - Add Compare and Branch Byte, 4-50
ACBD - Add Compare and Branch D_floating, 4-50
ACBF - Add Compare and Branch F_floating, 4-50
ACBG - Add Compare and Branch G_floating, 4-50
ACBH - Add Compare and Branch H_floating, 4-50
ACBL - Add Compare and Branch Long, 4-50
ACBW - Add Compare and Branch Word, 4-50
Accelerator
VAX-11/780, 9-15
Accelerator Control/Status Register (ACCS), 9-15
register (ACCS), 9-27
Accelerator Maintenance Register (ACCR), 9-15
Access across page boundaries, 5-13
Access control, 5-10
Access control violation fault, 6-17
Access mode, 6-5
memory, 6-5
Access mode, memory, 5-10
Executive, 5-10
Kernel, 5-10
Supervisor, 5-10
User, 5-10
Access type, operand, 3-2
address, 3-2, 3-18
branch, 3-2, 3-18
modify, 3-2, 3-18
synchronization, 3-19
read, 3-2, 3-18
write, 3-2, 3-18
ACCR - Accelerator Maintenance Register, 9-15
ACCS - Accelerator Control/Status register, 9-27
ACCS - Accelerator Control/Status Register, 9-15
ADAWI - Add Aligned Word

4-10
ADDB2 - Add Byte 2 Operand, 4-11
ADDB3 - Add Byte 3 Operand, 4-11
ADDD2 - Add D_floating 2 Operand, 4-122
ADDD3 - Add D_floating 3 Operand, 4-122
ADDF2 - Add F_floating 2 Operand, 4-122
ADDF3 - Add F_floating 3 Operand, 4-122
ADDG2 - ADDG_floating 2 Operand, 4-122
ADDG3 - ADDG_floating 3 Operand, 4-122
ADHH2 - ADD H_floating 2 Operand, 4-122
ADHH3 - ADD H_floating 3 Operand, 4-122
ADDL2 - Add Long 2 Operand, 4-11
ADDL3 - Add Long 3 Operand, 4-11
ADDP4 - Add Packed 4 Operand, 4-180
ADDP6 - Add Packed 6 Operand, 4-180
Address, 2-1
Address access type, operand, 3-2, 3-18
Address arguments, validating, 5-25
Address instructions, 4-38
Address translation, 5-6
Addressing modes notation, 3-4
ADDW2 - Add Word 2 Operand, 4-11
ADDW3 - Add Word 3 Operand, 4-11
ADWC - Add With Carry, 4-12
Alignment
stack, 6-35
target of control, 4-48
AO BLEQ - Add One and Branch
Less Than or Equal, 4-52
AO BLEQ - Add One and Branch
Less Than, 4-53
AP - Argument Pointer Register, 2-15
Argument Pointer Register, 2-15
Arithmetic faults, 6-14
Arithmetic instructions
decimal string, 4-175
floating point, 4-115
integer, 4-7
Arithmetic traps, 6-14
Array addressing, 3-14
ASHL - Arithmetic Shift Long, 4-13
ASHP - Arithmetic Shift and Round Packed, 4-182
ASHQ - Arithmetic Shift Quad, 4-13
AST - Asynchronous System Trap, 6-8, 6-33, 6-40
AST - Asynchronous System Trap, 6-34
AST, Asynchronous System Traps, 7-7
ASTLVL - Asynchronous System Trap Level, 6-8
ASTLVL - Asynchronous System Trap Level, 6-19, 6-39
ASTLVL - Pending AST Level, 7-5
Autodecrement addressing mode, 3-8
Autodecrement indexed addressing mode, 3-14
Autodecrement indexed mode, 3-14
Autodecrement mode, 3-8
Autoincrement addressing mode, 3-6
Autoincrement deferred addressing mode, 3-7
Autoincrement deferred indexed addressing mode, 3-14
mode, 3-14
Autoincrement deferred mode, 3-7
Autoincrement indexed addressing mode, 3-14
Autoincrement indexed mode, 3-14
Autoincrement mode, 3-6
Base operand specifier, 3-13
Base register, 2-15
BBC - Branch on Bit Clear, 4-56
BBCC - Branch on Bit Clear and Clear, 4-57
BBCCI - Branch on Bit Clear and Clear Interlocked, 4-59
BBCS - Branch on Bit Clear and Set, 4-57
BBS - Branch on Bit Set, 4-56
BBSC - Branch on Bit Set and Clear, 4-57
BBSS - Branch on Bit Set and Set, 4-57
BBSSI - Branch on Bit Set and Set Interlocked, 4-59
BCC - Branch on Carry Clear, 4-54
BRS - Branch on Carry Set, 4-54
BEQ - Branch on Equal, 4-54
BE QLU - Branch on Equal Unsigned, 4-54
BGEQ - Branch on Greater Than or Equal, 4-54
BGEQU - Branch on Greater Than or Equal Unsigned, 4-54
BGTR - Branch on Greater Than, 4-54
BGTRU - Branch on Greater Than Unsigned, 4-54
BICB2 - Bit Clear Byte 2 Operand, 4-14
BICB3 - Bit Clear Byte 3 Operand, 4-14
BICL2 - Bit Clear Long 2 Operand, 4-14
BICL3 - Bit Clear Long 3 Operand, 4-14
BICPSW - Bit Clear PSW, 4-79
BICW2 - Bit Clear Word 2 Operand, 4-14
BICW3 - Bit Clear Word 3 Operand, 4-14
BISB2 - Bit Set Byte 2 Operand, 4-15
BISB3 - Bit Set Byte 3 Operand, 4-15
BISL2 - Bit Set Long 2 Operand, 4-15
BISL3 - Bit Set Long 3 Operand, 4-15
BISPSW - Bit Set PSW, 4-80
BISW2 - Bit Set Word 2 Operand, 4-15
BISW3 - Bit Set Word 3 Operand,
Bit efficiency as a goal, 1-1
BITB - Bit Test Byte, 4-16
BITL - Bit Test Long, 4-16
BITW - Bit Test Word, 4-16
BLBC - Branch on Low Bit Clear, 4-61
BLBS - Branch on Low Bit Set, 4-61
BLEQ - Branch on Less Than or Equal, 4-54
BLEQU - Branch on Less Than or Equal Unsigned, 4-54
BLSS - Branch on Less Than, 4-54
BLSSU - Branch on Less Than Unsigned, 4-54
BNEQ - Branch on Not Equal, 4-54
BNEQU - Branch on Not Equal Unsigned, 4-54
BPT - Breakpoint Fault, 4-81
Braces as a notation, 3-4
Branch access type, operand, 3-2, 3-18
Branch displacement addressing, 3-17
BRB - Branch Byte Displacement, 4-62
Breakpoint fault, 6-21
BRW - Branch Word Displacement, 4-62
BSBB - Branch to Subroutine Byte Displacement, 4-63
BSBW - Branch to Subroutine Word Displacement, 4-63
Bugcheck, 4-227
BUGL, 4-227
BUGW, 4-227
BVC - Branch on Overflow Clear, 4-54
BVS - Branch on Overflow Set, 4-54
Byte, 2-1
Byte data type, operand, 3-2
Byte displacement addressing mode, 3-8
Byte displacement deferred addressing mode, 3-9
indexed addressing mode, 3-14
indexed mode, 3-14
Byte displacement deferred mode,
CLRD - Clear D floating, 4-124
CLRF - Clear F floating, 4-124
CLRG - Clear G floating, 4-124
CLRH - Clear H floating, 4-124
CLRL - Clear Long, 4-17
CLRO - Clear Octa, 4-17
CLRQ - Clear Quad, 4-17
CLRW - Clear Word, 4-17
CMI Error register, 9-24
CMIERR - CMI Error register, 9-24
CMP - Compatibility Mode, 6-5
CMPB - Compare Byte, 4-18
CPM3C - Compare Characters
  3 Operands, 4-148
CPMS - Compare Characters
  5 Operands, 4-148
CMPD - Compare D floating, 4-125
CMPP - Compare F floating, 4-125
CMPPG - Compare G floating, 4-125
CMPPH - Compare H floating, 4-125
CMPL - Compare Long, 4-18
CMPP3 - Compare Packed
  3 Operands, 4-184
CMPP4 - Compare Packed
  4 Operands, 4-184
CMPPV - Compare Field, 4-42
CMPPW - Compare Word, 4-18
CMPPZV - Compare Zero Extended Field, 4-42
Compatibility
  as a goal, 1-1
Compatibility (PDP-11)
  longword data format, 2-2
Compatibility mode, 6-5
  address modes, 10-2
  addresses, 10-54
  BPT fault, 10-57
  EMT fault, 10-57
  entering, 10-53
  exceptions, 10-57
  I/O, 10-61
  illegal instruction fault, 10-57
  instructions, 10-7
  interrupts, 10-57
  IOT fault, 10-57
  leaving, 10-53
  memory management, 10-54
  processor registers, 10-61
  PSW, 10-6
  register mapping, 10-53
  registers, 10-2
reserved instructions, 10-8
reserved instruction fault, 10-57
stack, 10-6
synchronization, 10-61
T-bit, 10-58
TRAP fault, 10-57
trap instructions, 10-8
unimplemented traps, 10-60
user environment, 10-2
Compatibility mode exception, 6-21
Condition Codes, 2-17, 6-5
Console Receive Control/Status register (RXCS), 9-8
Console Receive Data Buffer register (RXDB), 9-8
Console Storage
  Receive Status register (CSRS), 9-25
Console Storage Device, 9-25
Console Storage Receive
  Data Buffer register (CSRDB), 9-25
Console Storage Transmit
  Status register (CSTS), 9-25
Console Storage Transmit Data
  Buffer register (CSTD), 9-25
Console terminal registers, 9-8
Console Transmit Control/Status register (TXCS), 9-9
Console Transmit Data Buffer
  register (TXDB), 9-9
Constraints on I/O registers, 8-5
Context switching, 7-1
Context, process, 6-1, 6-3, 6-5, 6-34, 1-1 to 7-2
Context, system wide, 6-1, 6-34
Control instructions, 4-48
Control Store, Micro
  VAX-11/780, 9-17
Conventions
  general, 1-2
  in notation, 4-6
CRC - Calculate Cyclic
  Redundancy Check, 4-172
CSRD - Console Storage Receive
  Data Buffer register, 9-25
CSRS - Console Storage Receive
  Status register, 9-25
CSS, Reserved to, 1-3
CSTD - Console Storage Transmit
Data Buffer register, 9-25
CSTS - Console Storage Transmit Status register, 9-25
CUR MOD - Current Mode, 6-5
Currency sign, 4-205
Current Frame Pointer Register, 2-15
Current mode, 6-5
Customers, Reserved to, 1-3
CVTD - Convert Byte to D_floating, 4-126
CVTF - Convert Byte to F_floating, 4-126
CVTG - Convert Byte to G_floating, 4-126
CVTH - Convert Byte to H_floating, 4-126
CVTL - Convert Byte to Long, 4-126
CVTW - Convert Byte to Word, 4-126
CVTD - Convert D_floating to Byte, 4-126
CVTF - Convert D_floating to F_floating, 4-126
CVTD - Convert D_floating to H_floating, 4-126
CVTL - Convert D_floating to Long, 4-126
CVTDW - Convert D_floating to Word, 4-126
CVTF - Convert F_floating to Byte, 4-126
CVTF - Convert F_floating to D_floating, 4-126
CVTG - Convert F_floating to G_floating, 4-126
CVTH - Convert F_floating to H_floating, 4-126
CVTL - Convert F_floating to Long, 4-126
CVTFW - Convert F_floating to Word, 4-126
CVTB - Convert G_floating to Byte, 4-126
CVTF - Convert G_floating to D_floating, 4-126
CVTG - Convert G_floating to F_floating, 4-126
CVTH - Convert G_floating to H_floating, 4-126
CVTL - Convert G_floating to Long, 4-126
CVTGW - Convert G_floating to Word, 4-126
CVTHB - Convert H_floating to Byte, 4-126
CVTHD - Convert H_floating to D_floating, 4-126
CVTHF - Convert H_floating to F_floating, 4-126
CVTHG - Convert H_floating to G_floating, 4-126
CVTHL - Convert H_floating to Long, 4-126
CVTHW - Convert H_floating to Word, 4-126
CVTLB - Convert Long to Byte, 4-19
CVTLD - Convert Long to D_floating, 4-126
CVTLF - Convert Long to F_floating, 4-126
CVTLL - Convert Long to G_floating, 4-126
CVTLH - Convert Long to H_floating, 4-126
CVTL - Convert Long to Packed, 4-185
CVTLW - Convert Long to Word, 4-19
CVTPL - Convert Packed to Long, 4-187
CVTFS - Convert Packed to Leading Separate Numeric, 4-189
CVTP - Convert Packed to Trailing Numeric, 4-191
CVTRDL - Convert Rounded D_floating to Long, 4-126
CVTRFL - Convert Rounded F_floating to Long, 4-126
CVTRGL - Convert Rounded G_floating to Long, 4-126
CVTRHL - Convert Rounded H_floating to Long, 4-126
CVTSP - Convert Leading Separate Numeric to Packed, 4-193
CVTTP - Convert Trailing Numeric to Packed, 4-195
CVTWB - Convert Word to Byte, 4-19
CVTWD - Convert Word to D_floating, 4-126
CVTWF - Convert Word to F_floating, 4-126
CVTWG - Convert Word to
G_floating, 4-125
CVTWH - Convert Word to
H_floating, 4-126
CVTWL - Convert Word to Long,
4-19
Cyclic redundancy check, 4-171

D_floating, 2-4
D_floating data type,
operand, 3-2
Data sharing, 8-1
Data synchronization, 8-1
Data type
character string, 2-8
decimal string, 2-13
floating, 2-4 to 2-5
integer, 2-1 to 2-3
packed decimal string, 2-13
string, 2-8, 2-13
variable length bit field, 2-6
Data type, operand, 3-2
byte, 3-2
D_floating, 3-2
F_floating, 3-2
G_floating, 3-2
H_floating, 3-2
longword, 3-2
octword, 3-2
quadword, 3-2
word, 3-2
Data types, 2-1
DEC, Reserved to, 1-3
DECB - Decrement Byte, 4-20
Decimal overflow, 2-18, 6-5
Decimal string data type
packed, 2-13
Decimal string divide by
zero trap, 6-15
Decimal string instructions,
4-175
Decimal string overflow trap,
5-15
DECL - Decrement Long, 4-20
DECW - Decrement Word, 4-20
Digits
significant, 4-205
Dispatch
CHMx, 6-42
Displacement addressing mode,
3-9
Displacement deferred indexed
addressing mode, 3-14
mode, 3-14
Displacement mode, 3-9
DIVB2 - Divide Byte 2 Operand,
4-21
DIVB3 - Divide Byte 3 Operand,
4-21
DIVD2 - Divide D floating
2 Operand, 4-130
DIVD3 - Divide D floating
3 Operand, 4-130
DIVF2 - Divide F floating
2 Operand, 4-130
DIVF3 - Divide F floating
3 Operand, 4-130
DIVG2 - Divide G floating
2 Operand, 4-130
DIVG3 - Divide G floating
3 Operand, 4-130
DIVH2 - Divide H floating
2 Operand, 4-130
DIVH3 - Divide H floating
3 Operand, 4-130
Divide by zero fault, 6-16
Divide by zero trap, 5-15
DIVL2 - Divide Long 2 Operand,
4-21
DIVL3 - Divide Long 3 Operand,
4-21
DIVP - Divide Packed, 4-197
DIVW2 - Divide Word 2 Operand,
4-21
DIVW3 - Divide Word 3 Operand,
4-21
DV - Decimal Overflow Enable,
2-18, 6-5
Edit instruction, 4-205
EDITPC - Edit Packed to
Character String, 4-206
EDIV - Extended Divide, 4-23
Efficiency, bit
as a goal, 1-1
EMODD - Extended Multiply and
Integerize D floating, 4-132
EMODF - Extended Multiply and
Integerize F floating, 4-132
EMODG - Extended Multiply and
Integerize G floating, 4-132
EMODH - Extended Multiply and
Integerize H floating, 4-132
EMUL - Extended Multiply, 4-24
Entry mask, 4-70
EO$ADJUST_INPUT - Adjust Input
Length, 4-224
EOSBLANK_ZERO - Blank Backwards
   When Zero, 4-220
EOSCLEAR_SIGNIF - Clear
   Significance, 4-223
EOSEND - End Edit, 4-225
EOSEND_FLOAT - End Floating Sign,
   4-219
EOSFILL - Store Fill, 4-215
EOSFLOAT - Float Sign, 4-217
EOSINSERT - Insert Character,
   4-213
EOSLOAD_FILL - Load Fill
   Register, 4-222
EOSLOAD_MINUS - Load Sign
   Register If Minus, 4-222
EOSLOAD_PLUS - Load Sign
   Register If Plus, 4-222
EOSLOAD_SIGN - Load Sign
   Register, 4-222
EOSMOVE - Move Digits, 4-216
EOSREPLACE_SIGN - Replace Sign
   When Zero, 4-221
EOSSET_SIGNIF - Set Significance,
   4-223
EOSSTORE_SIGN - Store Sign,
   4-214
Errors, processor, 8-4
ESP - Executive Stack Pointer,
   7-4
Exception, 6-3
Exception condition, 6-1
Exceptions detected during
   operand reference, 6-18
   the operation, 6-14
Exceptions occurring as the
   consequence of an
   instruction, 6-20
Executive memory access mode,
   5-10
Extensibility
   as a goal, 1-1
Extension, specifier,
   3-9 to 3-10, 3-13
Extent, 1-2
EXTV - Extract Field, 4-44
EXTZEV - Extract Zero Extended
   Field, 4-44

Fault, 6-1, 6-3
   memory management, 5-23

Faults
   arithmetic, 6-14
   FF - Floating Fault Enable, 6-5
   FFC - Find First Clear, 4-45
   FFS - Find First Set, 4-45
   Field, 2-6
   FIELD - field addressing
   notation, 4-40
   Field instructions, 4-40
   Fill, 4-205
   Fill character, 4-205
   Fill register, 4-205
   First part done, 6-5
   Floating, 2-4 to 2-5
   Floating currency symbol, 4-205
   Floating data type, 2-4
   Floating divide by zero fault,
   6-16
   Floating divide by zero trap,
   6-15
   Floating fault, 6-5
   Floating overflow fault, 6-16
   Floating overflow trap, 6-15
   Floating point
   immediate constant, 3-12
   Floating point instructions,
   4-115
   Floating sign, 4-205
   Floating underflow, 2-18, 6-5
   Floating underflow fault, 6-16
   Floating underflow trap, 6-15
   FP - Current Frame Pointer
   Register, 2-15
   FPD - First Part Done, 6-5
   Frame Pointer Register, Current,
   2-15
   FU - Floating Underflow Enable,
   2-18, 6-5

G_floating, 2-5
G_floating data type,
   operand, 3-2
General mode addressing, 3-5
General Registers, 7-4
Global page table index (gptx),
   5-8
Goals, 1-1
Gptx, 5-8

H_floating, 2-5
H_floating data type,
   operand, 3-2
HALT - Halt, 4-82
Halt, processor, 6-26, 6-28, 6-34, 6-38, 6-41, 6-43, 8-2, 9-18
VAX-11/780, 6-29

I/O instructions, A-18
I/O structure, 2-20, 8-5
ICCS - Interval Clock
  Control/Status register, 9-13
ICR - Interval Count Register, 9-13
Immediate addressing mode, 3-6
Immediate constant
  floating point, 3-12
  integer, 3-11
Immediate indexed
  addressing mode, 3-14
Immediate indexed mode, 3-14
Immediate mode, 3-6
INCB - Increment Byte, 4-25
INCL - Increment Long, 4-25
INCW - Increment Word, 4-25
INDEX - Compute Index, 4-83
Index addressing mode, 3-13
Index mode, 3-13
Index register, 2-15
Indivisible operation
  modify access, 3-19
Initialize UNIBUS (IORESET), 9-27
Initiate exception or interrupt, 6-37
INSQHI - Insert Entry into Queue at Head, Interlocked, 4-99
INSQTI - Insert Entry into Queue at Tail, Interlocked, 4-102
INSQUE - Insert Entry in Queue, 4-105
Instruction format, 2-19
Instruction operand formats, A-1
INSV - Insert Field, 4-47
Integer
  immediate constant, 3-11
  integer data type, 2-1 to 2-3
  integer divide by zero trap, 6-15
Integer instructions, 4-7
Integer overflow, 2-18, 5-5
Integer overflow trap, 5-14
Interrupt, 6-1 to 6-3, 6-8
Interrupt AST Delivery, 7-8
Interrupt priority level, 6-5
Interrupt Priority Level (IPL), 6-2, 6-11
Interrupt process, 6-8
Interrupt stack, 5-5
Interrupt stack not valid halt, 6-26
Interrupt structure, 2-20
Interrupt, Process Scheduling, 7-8
Interrupts, 8-4
Interrupts, Process Structure, 7-8
Interval clock, 9-13
Interval Clock Control/Status register (ICCS), 9-13
Interval Count Register (ICR), 9-13
IORESET - Initialize UNIBUS, 9-27
IPL - Interrupt Priority Level, 6-2, 6-5, 6-10 to 6-11
IS - Interrupt Stack in use, 6-5, 6-35
IV - Integer Overflow Enable, 2-18, 6-5
JMP - Jump, 4-65
JSB - Jump To Subroutine, 4-66
Kernel memory access mode, 5-10
Kernel stack not valid abort, 6-26
KSP - Kernel Stack Pointer, 7-4
LDPCTX - Load Process Context, 7-9
LDPCTX -- Load Process Context, 4-226
Leading separate sign, 4-175, 4-189, 4-193
Leading zero, 4-223
Literal addressing mode, 3-11
Literal mode, 3-11
LOCC - Locate Character, 4-152
Logical instructions, 4-7
Longword, 2-2
  PDP-11 compatibility, 2-2
Longword data type, operand, 3-2
Longword displacement
  addressing mode, 3-8
Longword displacement deferred
  addressing mode, 3-9
  indexed addressing mode,
M - Modify bit, 5-6
Machine Check Error Summary register (MCESR), 9-26
Machine check exception, 6-26
MAPEN - Map Enable Register, 5-5
MATCHC - Match Characters, 4-154
MBRK - Micro Program Breakpoint Address register, 9-18
MBZ, 1-2
MCESR - Machine Check Error Summary register, 9-25
MCOMB - Move Complemented Byte, 4-26
MCOML - Move Complemented Long, 4-26
MCOMW - Move Complemented Word, 4-26
Memory access mode, 5-10, 6-5
Executive, 5-10
Kernel, 5-10
Supervisor, 5-10
User, 5-10
Memory management control, 5-5
Memory management exceptions, 6-17
Memory management faults, 5-23
Memory Mapping Enable (MAPEN), 5-5
MPFR - Move From Processor Register, 9-5
MPFR -- Move From Processor Register, 4-226
Micro Control Store
VAX-11/780, 9-17
Micro Program Breakpoint Address register (MBRK), 9-18
MINU - minimum unsigned notation, 4-6
Miscellaneous instructions, 4-78
MME - Memory Mapping Enable, 5-5
MNEG - Move Negated Byte, 4-27
MNEG - Move Negated D_floating, 4-134
MNEG - Move Negated F_floating, 4-134
MNEG - Move Negated H_floating, 4-134
MNEG - Move Negated Long, 4-27
MNEG - Move Negated Word, 4-27
Mode, 5-10, 6-5
compatibility, 6-5
Mode changing instructions, 6-41
Mode, memory access, 5-10, 6-5
Modify access type, operand, 3-2, 3-18
synchronization, 3-19
Modify bit, 5-6
MOVAB - Move Address Byte, 4-38
MOVAD - Move Address D_floating, 4-38
MOVAF - Move Address F_floating, 4-38
MOVAG - Move Address G_floating, 4-38
MOVAH - Move Address H_floating, 4-38
MOVAL - Move Address Long, 4-38
MOVAO - Move Address Octa, 4-38
MOVAQ - Move Address Quad, 4-38
MOVAW - Move Address Word, 4-38
MOVB - Move Byte, 4-28
MOVC - Move Character 3 Operand, 4-156
MOVC - Move Character 5 Operand, 4-156
MOVD - Move D_floating, 4-135
MOVF - Move F_floating, 4-135
MOVG - Move G_floating, 4-135
MOVH - Move H_floating, 4-135
MOVL - Move Long, 4-28
MOVO - Move Octa, 4-28
MOVP - Move Packed, 4-199
MOVPSL - Move PSL, 4-85
MOVQ - Move Quad, 4-28
MOVTC - Move Translated Characters, 4-160
MOVTC - Move Translated Until Character, 4-163
MOVW - Move Word, 4-28
MOVZBL - Move Zero-Extended Byte to Long, 4-29
MOVZBW - Move Zero-Extended Byte to Word, 4-29
MOVZWL - Move Zero-Extended Word to Long, 4-29
MTPR - Move To
Processor Register, 9-3
MTPR -- Move To Processor Register, 4-226
MULB2 - Multiply Byte 2 Operand, 4-30
MULB3 - Multiply Byte 3 Operand, 4-30
MULD2 - Multiply D floating 2 Operand, 4-136
MULD3 - Multiply D floating 3 Operand, 4-136
MULF2 - Multiply F floating 2 Operand, 4-136
MULF3 - Multiply F floating 3 Operand, 4-136
MULG2 - Multiply G floating 2 Operand, 4-136
MULG3 - Multiply G floating 3 Operand, 4-136
MULH2 - Multiply H floating 2 Operand, 4-136
MULH3 - Multiply H floating 3 Operand, 4-136
MUL2L - Multiply Long 2 Operand, 4-30
MUL3L - Multiply Long 3 Operand, 4-30
MULP - Multiply Packed, 4-201
MULW2 - Multiply Word 2 Operand, 4-30
MULW3 - Multiply Word 3 Operand, 4-30

R[n], 2-15
register, 2-15
REM - remainder, 4-6
Rn, 2-15
SEXT - sign extend, 3-4, 4-6
ZEXT - zero extend, 3-4, 4-6
Numbering, 1-2

OA - operand address notation, 3-4
Octaword, 2-3
Octaword data type, operand, 3-2
Opcode assignments, A-12
Opcode formats, 3-1
Opcode reserved to customers
fault, 6-20
Opcode reserved to DIGITAL Fault, 6-20
Operand format summary, A-1
Operand specifier, 3-2
Operand specifier access type, 3-2
Operand specifier conventions, 3-18
Operand specifier data type, 3-2
Operand specifier notation, A-9
Operand specifier, base, 3-13
Operand, primary, 3-13
Orthogonality
as a goal, 1-1
Overflow, 6-4 to 6-5,
6-14 to 6-16, 6-27
stack, 6-26

P0 Base Register, 7-4
P0 Base Register (P0BR), 5-17
P0 Length Register (P0LR), 5-17
P0 Limit Register, 7-4
P0 Page Table (P0PT), 5-17
P0 Region, 5-17
P0 region, 5-4
P0BR - P0 Base Register, 5-17, 7-4
P0LR - P0 Length Register, 5-17
P0LR - P0 Limit Register, 7-4
P0PT - P0 Page Table, 5-17
P1 Base Register, 7-5
P1 Base Register (P1BR), 5-20
P1 Length Register (P1LR), 5-20
P1 Limit Register, 7-5
P1 Page Table (P1PT), 5-20
P1 Region, 5-20
P1 region, 5-4

N - Negative Condition Code, 2-17, 6-5
N condition code, 2-17, 6-5
Next Interval Count Register (NICR), 9-13
Nibble, 2-13
NICR - Next Interval Count Register, 9-13
NOP - No Operation, 4-86
as a diagnostic scope point, 9-18
Notation
(), 3-4
{}, 3-4
addressing modes, 3-4
FIELD - field addressing, 4-40
MINU - minimum unsigned, 4-6
OA - operand address, 3-4
operand specifier, 4-3, A-9
operation description, 4-4
PIBR - P1 Base Register, 5-20, 7-5
PLL R - P1 Length Register, 5-20
PLLR - P1 Limit Register, 7-5
PLPT - P1 Page Table, 5-20
Packed decimal
instructions, 4-175
Packed decimal string, 2-13
Page, 5-2
Page frame number field, 5-6
Page Table Entry (PTE), 5-6, 5-8
Parentheses
as a notation, 3-4
Part done, 6-5
PC - Program Counter Register, 2-15
in process context, 7-4
PCB - Process Control Block, 7-2
PCBB - Process Control Block Base, 7-2
Per-process Space, 5-4
Performance monitor enable, 7-5
PFN - Page Frame Number field, 5-6
PME - Performance Monitor Enable, 7-5
POLYD - Polynomial Evaluation
D_floating, 4-138
POLYP - Polynomial Evaluation
P_floating, 4-138
POLYG - Polynomial Evaluation
G_floating, 4-138
POLYH - Polynomial Evaluation
H_floating, 4-138
POPR - Pop Registers, 4-87
Power fail, 8-2
Previous mode, 6-5
Primary operand, 3-13
Priority level, 6-5
Probe accessibility, 5-26, 5-28
PROBER - Probe Read
Accessibility, 4-226
accessibility, 5-26
PROBEW - Probe Write
Accessibility, 4-226
accessibility, 5-26
Procedure call instructions, 4-70
Procedure calling interface, 4-70
Process context, 7-1
Process control block, 7-2
Process scheduling, 7-1
Process Space, 5-4, 5-16
Process, definition, 7-1
Processor Errors, 8-4
Processor Internal Register
space, 9-1
Processor mode, 5-10
Processor Registers, 9-6
Processor Status Longword (PSL), 6-5
Processor Status Word, 2-17
Processor type, 9-7
Program counter
in process context, 7-4
Program Counter Register, 2-15
Program status longword
in process context, 7-4
PROT - Protection field, 5-6
Protection, 5-10
check, 4-222
Protection Code, 5-10
Protection field, 5-6
PRV_MOD - Previous Mode, 6-5
PSL - Processor Status Longword, 6-5
PSL - Program Status Longword
in process context, 7-4
PSW - Processor Status Word,
2-17, 6-3, 6-5, 6-19
PTE - Page Table Entry, 5-6, 5-8
PUSHAB - Push Address Byte, 4-39
PUSHAD - Push Address D_floating, 4-39
PUSHAF - Push Address F_floating, 4-39
PUSHAG - Push Address G_floating, 4-39
PUSHAH - Push Address H_floating, 4-39
PUSHAL - Push Address Long, 4-39
PUSHAQ - Push Address Quad, 4-39
PUSHAW - Push Address Word, 4-39
PUSHL - Push Long, 4-31
PUSHR - Push Registers, 4-88
Quadword, 2-3
Quadword data type, operand, 3-2
Queue instructions, 4-98
Range
as a goal, 1-2
Range of values, 1-2
Read access type, operand, 3-2, 3-18
Register
fill, 4-205
sign, 4-205
Register addressing mode,
3-5 to 3-6
Register deferred
addressing mode, 3-5
Register deferred indexed
addressing mode, 3-14
Register deferred indexed mode,
3-14
Register deferred mode, 3-5
Register mode, 3-5 to 3-6
Register usage, 2-15
Registers
VAX-11 Series, 9-6
VAX-11/750 Specific, 9-24
VAX-11/780 Specific, 9-15
REI - Return from Exception
or Interrupt, 6-39
REI -- Return from Exception
or Interrupt, 4-226
REM - remainder notation, 4-6
REMQHI - Remove Entry from Queue
at Head, Interlocked, 4-107
REMQTI - Remove Entry from Queue
at Tail, Interlocked, 4-110
REMQUE - Remove Entry from Queue,
4-113
RESERVED, 1-3
Reserved addressing mode fault,
6-18
Reserved operand exception, 6-18
Restartability, 8-3
RET - Return From Procedure,
4-76
Revision level, 9-7
ROTL - Rotate Long, 4-32
RSB - Return From Subroutine,
4-67
RXCS - Console Receive
Control/Status register, 9-8
RXDB - Console Receive
Data Buffer register, 9-8
Saved PC, 6-3, 6-5, 6-14, 6-18,
6-22, 6-27 to 6-28
Saved PSL, 6-3, 6-5, 6-14,
6-21 to 6-23, 6-25,
6-27 to 6-28
Saved TP, 6-22 to 6-23, 6-25,
6-27 to 6-28
SBI Error register (SBIER), 9-21
SBI Fault/Status register (SBIFS),
9-18
SBI Maintenance register (SBIMT),
9-20
SBI Quad Clear (SBIQC), 9-23
SBI Silo Comparator
register (SBISC), 9-19
SBI Silo Data Register (SBIS),
9-19
SBI Timeout Address
register (SBITA), 9-22
SBIER - SBI Error register, 9-21
SBIFS - SBI Fault/Status register,
9-18
SBIMT - SBI Maintenance register,
9-20
SBIQC - SBI Quad Clear, 9-23
SBIS - SBI Silo Data Register,
9-19
SBISC - SBI Silo Comparator
register, 9-19
SBITA - SBI Timeout Address
register, 9-22
SBR - System Base Register, 5-13
SBWC - Subtract With Carry, 4-33
SCANC - Scan Characters, 4-165
SCBB - System Control Block Base,
6-29
Scheduling, process, 7-1
Self-relative queues, 4-95
Separate sign, leading, 4-175,
4-189, 4-193
Separation of procedure and data,
2-20
Serial number, 9-7
Serialization of notification
of multiple events, 6-27
SEXT - sign extend notation,
3-4, 4-6
Sharing, 8-1
SID - System Identification, 9-7
Sign, 4-205
currency, 4-205
Sign character, 4-205
Sign register, 4-205
Significance, 4-205
Significance indicator, 4-205,
4-223
Significant digits, 4-205
SIRR - Software Interrupt
Request Register, 6-2, 6-8,
6-10 to 6-11
SISR - Software Interrupt
Summary Register, 6-10
SKPC - Skip Character, 4-167
SLR - System Length Register, 5-13
SOBGEQ - Subtract One and Branch Greater Than or Equal, 4-68
SOBGTR - Subtract One and Branch Greater Than, 4-69
Software Interrupt
Request Register (SIRR), 6-10
Summary Register (SISR), 6-10
Software interrupt, 6-10
SP - Stack Pointer Register, 2-15
SPANC - Span Characters, 4-169
Specifier extension, 3-9 to 3-10, 3-13
SPT - System Page Table, 5-13
SSP - Supervisor Stack Pointer, 7-4
Stack alignment, 6-35
Stack frame, 4-70
Stack pointer
in process context, 7-4
Stack pointer images, 9-2
Stack Pointer Register, 2-15
Stack residency, 6-34
Stack, switch, 6-34, 6-37, 6-39
String data type
character, 2-8
packed decimal, 2-13
String descriptor
as operand, 4-146, 4-175
String instructions
cyclic redundancy check, 4-171
decimal, 4-175
SUBB2 - Subtract Byte 2 Operand, 4-34
SUBB3 - Subtract Byte 3 Operand, 4-34
SUBD2 - Subtract D_floating
2 Operand, 4-143
SUBD3 - Subtract D_floating
3 Operand, 4-143
SUBF2 - Subtract F_floating
2 Operand, 4-143
SUBF3 - Subtract F_floating
3 Operand, 4-143
SUBG2 - Subtract G_floating
2 Operand, 4-143
SUBG3 - Subtract G_floating
3 Operand, 4-143
SUBH2 - Subtract H_floating
2 Operand, 4-143
SUBH3 - Subtract H_floating
3 Operand, 4-143
SUBL2 - Subtract Long 2 Operand, 4-34
SUBL3 - Subtract Long 3 Operand, 4-34
SUBP4 - Subtract Packed
4 Operand, 4-203
SUBP6 - Subtract Packed
6 Operand, 4-203
Subscript range trap, 6-16
SUBW2 - Subtract Word 2 Operand, 4-34
SUBW3 - Subtract Word 3 Operand, 4-34
Summary, 1-1
Supervisor memory access mode, 5-10
SVPCTX - Save Process Context, 7-11
SVPCTX -- Save Process Context, 4-226
Switching context, 7-1
Synchronization, 8-1
modify access, 3-19
System Base Register (SBR), 5-13
System Control Block Base (SCBB), 6-29
System Identification register (SID), 9-7
System Length Register (SLR), 5-13
System Page Table (SPT), 5-13
System Region, 5-13
System Space, 5-4, 5-13
T - Trace Enable, 6-5
T - Trace Trap Enable, 2-18
TB Data register (TBDATA), 9-28
TB Group Disable register (TBDR), 9-26
TBCHK - Translation Buffer Check register, 5-23
TBDATA - TB Data Register, 9-28
TBDR - TB Group Disable register, 9-26
TBIA - Translation Buffer Invalidate All Register, 5-23
TBIS - Translation Buffer
  Invalidate Single Register, 5-22
Terminology
general, 1-2
Time-of-Year Register (TODR), 9-11
TODR - Time-of-Year Register, 9-11
TP - Trace Pending, 6-5
Trace, 6-5, 6-22
Trace pending, 6-5
Trace trap, 2-18
Trailing numeric
  string instructions, 4-175
  string instructions, 4-191,
  4-195
Translation buffer, 5-22
Translation Buffer Check
  register (TBCHK), 5-23
Translation Buffer Invalidate
  All Register (TBIA), 5-23
  Single Register (TBIS), 5-22
Translation not valid fault,
  6-17
Translation, address, 5-6
Trap, 6-1, 6-3
Traps
  arithmetic, 6-14
TSTB - Test Byte, 4-35
TSTD - Test D_float, 4-145
TSTF - Test F_float, 4-145
TSTG - Test G_float, 4-145
TSTH - Test H_float, 4-145
TSTL - Test Long, 4-35
TSTW - Test Word, 4-35
TXCS - Console Transmit
  Control/Status register, 9-9
TXDB - Console Transmit
  Data Buffer register, 9-9
Type, processor, 9-7
UNDEFINED, 1-2
UNIBUS, 6-2, 6-9, 8-1
Unmapped system, 5-5
UNPREDICTABLE, 1-2
Unsigned integer, 2-1 to 2-2
User memory access mode, 5-10
USP - User Stack Pointer, 7-4
V - Overflow Condition Code,
  2-17, 6-5
V - Valid bit, 5-6
V condition code, 2-17, 6-5
Valid bit, 5-6
Validating address arguments,
  5-25
Variable length bit field
  instructions, 4-40
  bytes referenced, 2-7
  data type, 2-6
VAX-11/780 Accelerator, 9-15
VAX-11/780 Micro Control Store,
  9-17
Vector, 6-2, 6-20 to 6-21,
  6-26 to 6-27, 6-29, 6-31,
  6-34 to 6-35, 6-37, 6-42
  interrupt, 6-8
Virtual address, 2-1
Virtual Address Space, 5-2
Virtual Page Number, 5-4
VPN - Virtual Page Number, 5-4
WCSA - Writable Control Store
  Address register, 9-17
WCSD - Writable Control Store
  Data register, 9-17
Word, 2-2
Word data type, operand, 3-2
Word displacement
  addressing mode, 3-8
  Word displacement deferred
  addressing mode, 3-9
    indexed addressing mode,
    3-14
    indexed mode, 3-14
Word displacement deferred mode,
  3-9
Word displacement indexed
  addressing mode, 3-14
Word displacement indexed mode,
  3-14
Word displacement mode, 3-8
Writable Control Store Address
  register (WCSA), 9-17
Writable Control Store Data
  register (WCSD), 9-17
Write access type, operand,
  3-2, 3-18
XFC - Extended Function Call,
  4-89
XORB2 - Exclusive OR Byte
  2 Operand, 4-36
XORB3 - Exclusive OR Byte
  3 Operand, 4-36
XORL2 - Exclusive OR Long
2 Operand, 4-36
XORL3 - Exclusive OR Long
3 Operand, 4-36
XORW2 - Exclusive OR Word
2 Operand, 4-36
XORW3 - Exclusive OR Word
3 Operand, 4-36

Z - Zero Condition Code, 2-17, 6-5
Z condition code, 2-17, 6-5
Zero
leading, 4-223
ZEXT - zero extend notation, 3-4, 4-5
Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our manuals.

What is your general reaction to this manual? (format, accuracy, completeness, organization, etc.)

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

What features are most useful?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

Does the publication satisfy your needs?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

What errors have you found?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

Additional comments

________________________________________________________________________
________________________________________________________________________

Name

________________________________________________________________________

Title

________________________________________________________________________

Company

________________________________________________________________________

Dept.

Address

________________________________________________________________________

City

________________________________________________________________________

State

Zip