EL-00032-00

DEC STD 032
VAX Architecture
Standard

digital™

digital

INTEROFFICE MEMORANDUM

DATE: 15-Jan-1990
DEPT: Standards & Methods
EXT : 287-3724
LOC/MS : CTS1-2/D4
ENET: JOKUR::SMC

To: Holders of A-DS-EL00032-00
Holders of Complete Sets of DEC Standards

Subj: Document Revision

Enclosed is a new update to DEC STD 032, A-DS-EL00032-00-0. This document has been completely revised, so please remove the text and tabs from your current manual and replace with the new enclosed material.

If you require more information or wish to be removed from distribution
please contact Standards & Methods control at DTN 287-3724 or JOKUR::SMC.

If you no longer require the enclosed information, please return it along with your name, badge number and location so that we may remove your name from our distribution list.

DOCUMENT IDENTIFIER	A-DS-EL00032-00-0 REV J
DOCUMENT TITLE	DEC STD 032 VAX ARCHITECTURE STANDARD

digital™

```
EEEEEE    CCCC     0000
E        C    C   0    0
EEEE     C        0    0
E        C    C   0    0
EEEEEE    CCCC     0000
```

ECO No: EL00032-00-CTS08

| d | i | g | i | t | a | l | TM

ENGINEERING CHANGE ORDER

Orig.:	Richard Brunner	Sheet: 1 of 2	
Phone:	293-5364	Date Received:	15-Dec-1989
Loc.:	BXB1-1	MS: E11	
Cost Center No.: 31L		Final Issue: 15-Dec-1989	

Unit(s) to be changed: EL-00032-00

Prod. Families Affected: N/A

ECO Type Code:	5	Category:	Standard	FCC Req.:	No
F/S Affected:	No	FCO Req.:	No	LOU Code:	N/A
Where Used:	A-DS-EL00032-00-0				

Documentation Affected: A-DS-EL00032-00-0, DEC STD 032-0 VAX ARCHITECTURE STANDARD

PROBLEM Material needed updating.

CORRECTION Update the standard per sheet 2 of this ECO.

BREAK-IN/EFFECTIVITY Immediately.

APPROVAL NAMES		C/C	Phone No. DTN-Ext.	Date dd-mmm-yyyy
Engineering:	Richard A. Brunner Richard Brunner	31L	293-5364	15-Dec-1989
Standards Process Manager:	Eric A Williams Eric Williams	396	287-3696	15-Dec-1989
Coordinator:	Barbara Bowers (DF) Barbara Bowers	3VQ	287-3674	15-Dec-1989

Charge Number: 098-08511 W.O. No: 192947

digital™

```
EEEEEE   CCCC    0000
E       C    C  0    0
EEEE    C       0    0
E       C    C  0    0
EEEEEE   CCCC    0000
```

ECO No: EL00032-00-CTS08

d	i	g	i	t	a	l

TM

ENGINEERING CHANGE ORDER

CONTINUATION SHEET

Sheet: 2 of 2

ITEM	PART NUMBER	DOCUMENT NUMBER	OLD	NEW
NO.			REV	REV
1	EL-00032-00	A-DS-EL00032-00-0	H	J

DESCRIPTION:

- Removed the SPTEP register and the ability to have system page tables in virtual memory

- Significantly clarified the sharing of memory across multiple processors and I/O devices

- Expanded the discussion of instruction-stream coherency

- Added the description of the VAX Vector processor

- Added Address Space Numbers

- Clarified the possible physical address modes and what processor features each mode allows

DEC STD 032-0 VAX ARCHITECTURE STANDARD

DOCUMENT IDENTIFIER: A-DS-EL00032-00-0 Rev J, December 15, 1989

ABSTRACT: The VAX Architecture Standard is the definition of the VAX architecture. It specifies the operations provided by all VAX processors, and specifies constraints on software intended to run on VAX processors.

APPLICABILITY: This standard applies to all software written to run on VAX processors, all VAX processors, and all "closely coupled" VAX peripherals.

STATUS: Approved 15-Dec-1989; use VTX SMC for current status.

TITLE: DEC STD 032-0 VAX ARCHITECTURE STANDARD

DOCUMENT IDENTIFIER: A-DS-EL00032-00-0 Rev J, December 15, 1989

REVISION HISTORY:

Rev A,	10-Jul-1980		
Rev B,	01-Sep-1982		
Rev C,	15-Nov-1983		
Rev D,	26-Mar-1985		
Rev D1,	15-May-1986	ECO#	LJ004
Rev E,	20-Oct-1986	ECO#	LJ005
Rev F,	21-Nov-1986	ECO#	LJ006
Rev H,	19-Jun-1987	ECO#	LJ007
Rev J,	15-Dec-1989	ECO#	CTS08

Document Management Group: VAX System Architecture and Interconnect (SHV)
Responsible Department: Systems Architecture Group
Responsible Person: Richard A. Brunner

APPROVAL: This document has been reviewed and approved by the VAX architect and by the manager of the Systems Architecture Group.

Richard A. Brunner

Richard A. Brunner, VAX Architect

Audrey R Reith

Audrey Reith, Systems Architecture Manager

Eric A Williams

Eric Williams, Standards Process Manager

Questions about the VAX architecture can be answered in the EAGLE1::VAX notes conference or by contacting:

Rich Brunner
BXB1-1/E11
EAGLE1::SRM

Use VTX SMC to order copies of this document from Standards and Methods Control. Send distribution questions to JOKUR::SMC or call DTN: 287-3724.

A variant of this document is available to customers:

VAX Architecture Reference Manual
Order number EY-3459E-DP
Digital Press
Digital Equipment Corporation
12 Crosby Drive
Bedford MA 12730

CONTENTS

PREFACE

CHAPTER 1 BASIC ARCHITECTURE

1.1 ADDRESSING 1-3
1.2 DATA TYPES 1-3
1.3 PROCESSOR STATE 1-22

CHAPTER 2 INSTRUCTION FORMATS AND ADDRESSING MODES

2.1 OPCODE FORMATS 2-1
2.2 OPERAND SPECIFIERS 2-3
2.3 NOTATION 2-4
2.4 GENERAL MODE ADDRESSING FORMATS 2-7
2.5 BRANCH MODE ADDRESSING FORMATS 2-15
2.6 INSTRUCTION INTERPRETATION 2-18

CHAPTER 3 INSTRUCTIONS

3.1 NOTATION 3-2
3.2 INTEGER-ARITHMETIC AND LOGICAL INSTRUCTIONS . . . 3-7
3.3 ADDRESS INSTRUCTIONS 3-37
3.4 BIT-FIELD INSTRUCTIONS 3-41
3.5 CONTROL INSTRUCTIONS 3-49
3.6 PROCEDURE-CALL INSTRUCTIONS 3-71
3.7 MISCELLANEOUS INSTRUCTIONS 3-81
3.8 QUEUE INSTRUCTIONS 3-95
3.9 FLOATING-POINT INSTRUCTIONS 3-115
3.10 CHARACTER-STRING INSTRUCTIONS 3-151
3.11 CYCLIC-REDUNDANCY-CHECK INSTRUCTION 3-177
3.12 DECIMAL-STRING INSTRUCTIONS 3-183
3.13 EDIT INSTRUCTION 3-215

CHAPTER 4 MEMORY MANAGEMENT

4.1 VIRTUAL ADDRESS SPACE 4-2
4.2 MEMORY MANAGEMENT CONTROL 4-5
4.3 ADDRESS TRANSLATION 4-8
4.4 MEMORY PROTECTION 4-12
4.5 SYSTEM-SPACE ADDRESS TRANSLATION 4-16
4.6 PROCESS-SPACE ADDRESS TRANSLATION 4-21
4.7 TRANSLATION BUFFER 4-27
4.8 FAULTS AND PARAMETERS 4-30
4.9 PRIVILEGED SERVICES AND ARGUMENT VALIDATION . . 4-33

CHAPTER 5 INTERRUPTS AND EXCEPTIONS

5.1 PROCESSOR STATUS 5-5
5.2 RESTARTABILITY 5-5
5.3 INTERRUPTS 5-6
5.4 EXCEPTIONS 5-11

5.5 SERIALIZATION OF NOTIFICATION OF MULTIPLE EVENTS 5-21
5.6 SYSTEM CONTROL BLOCK 5-22
5.7 STACKS 5-27
5.8 INITIATE EXCEPTION OR INTERRUPT 5-31
5.9 INSTRUCTIONS RELATED TO EXCEPTIONS AND INTERRUPTS 5-37

CHAPTER 6 PROCESS STRUCTURE

6.1 PROCESS CONTEXT 6-1
6.2 ASYNCHRONOUS SYSTEM TRAPS 6-5
6.3 PROCESS-SCHEDULING INTERRUPTS 6-6
6.4 PROCESS-STRUCTURE INSTRUCTIONS 6-7

CHAPTER 7 MEMORY AND I/O

7.1 MEMORY, MULTIPROCESSING, AND INTERPROCESSOR COMMUNICATION. 7-1
7.2 INSTRUCTION STREAM PROCESSING AND MODIFICATION . . 7-9
7.3 MEMORY REFERENCES 7-14
7.4 CACHE 7-17
7.5 I/O STRUCTURE 7-18

CHAPTER 8 PRIVILEGED REGISTERS

8.1 PER-PROCESS REGISTERS AND CONTEXT SWITCHING . . . 8-1
8.2 STACK-POINTER IMAGES 8-2
8.3 MTPR AND MFPR INSTRUCTIONS 8-3

CHAPTER 9 PDP-11 COMPATIBILITY MODE

9.1 GENERAL REGISTERS AND ADDRESSING MODES 9-2
9.2 THE STACK 9-6
9.3 PROCESSOR STATUS WORD 9-6
9.4 INSTRUCTIONS 9-6
9.5 ENTERING AND LEAVING COMPATIBILITY MODE 9-53
9.6 NATIVE-MODE AND COMPATIBILITY-MODE REGISTERS . . 9-53
9.7 COMPATIBILITY-MODE MEMORY MANAGEMENT 9-53
9.8 COMPATIBILITY-MODE EXCEPTIONS AND INTERRUPTS . . 9-54
9.9 TRACING IN COMPATIBILITY MODE 9-55
9.10 UNIMPLEMENTED PDP-11 TRAPS 9-56
9.11 COMPATIBILITY-MODE I/O REFERENCES 9-57
9.12 PROCESSOR REGISTERS 9-57
9.13 PROGRAM SYNCHRONIZATION 9-57

CHAPTER 10 SYSTEM BOOTSTRAPPING AND CONSOLE

10.1 SYSTEM BOOTSTRAPPING 10-1
10.2 SYSTEM RESTART 10-3
10.3 SYSTEM POWERFAIL AND RECOVERY 10-7
10.4 MAJOR SYSTEM STATE TRANSITIONS 10-9
10.5 SYSTEM HALTED (CONSOLE I/O MODE) 10-10
10.6 SYSTEM RUNNING (PROGRAM I/O MODE) 10-32

CHAPTER 11 IMPLEMENTATION OPTIONS

11.1 INSTRUCTION-SET OPTIONS 11-1
11.2 OTHER IMPLEMENTATION OPTIONS 11-4
11.3 INSTRUCTION EMULATION 11-11

CHAPTER 12 VIRTUAL MACHINES

12.1 VIRTUAL-MACHINE STATE 12-5
12.2 INSTRUCTIONS TO SUPPORT VIRTUAL MACHINES 12-6
12.3 EMULATION OF VIRTUAL-MACHINE INSTRUCTIONS . . . 12-13
12.4 RING COMPRESSION 12-16
12.5 DIFFERENCES BETWEEN VIRTUAL VAXES AND REAL VAXES 12-18
12.6 COMPATIBILITY MODE 12-19
12.7 TIME ON VIRTUAL MACHINES 12-19

CHAPTER 13 VAX VECTOR REGISTERS AND INSTRUCTIONS

13.1 INTRODUCTION 13-1
13.2 REGISTERS 13-2
13.3 VECTOR INSTRUCTION FORMATS 13-12
13.4 VECTOR INSTRUCTION CLASSES 13-22
13.5 VECTOR MEMORY ACCESS INSTRUCTIONS 13-23
13.6 VECTOR INTEGER INSTRUCTIONS 13-33
13.7 VECTOR LOGICAL AND SHIFT INSTRUCTIONS 13-39
13.8 VECTOR FLOATING-POINT INSTRUCTIONS 13-43
13.9 VECTOR EDIT INSTRUCTIONS 13-57
13.10 MISCELLANEOUS INSTRUCTIONS 13-60
13.11 ASSEMBLER NOTATION 13-65

CHAPTER 14 VAX VECTOR EXECUTION MODEL AND EXCEPTION REPORTING

14.1 EXECUTION MODEL 14-1
14.2 VECTOR PROCESSOR EXCEPTIONS 14-15
14.3 SYNCHRONIZATION 14-29
14.4 MEMORY MANAGEMENT 14-41
14.5 HARDWARE ERRORS 14-43

APPENDIX A OPCODE ASSIGNMENTS

APPENDIX B IMPLEMENTATION DEPENDENCIES

B.1 IMPLEMENTATION OPTIONS B-1
B.2 MICROVAX I B-7
B.3 MICROVAX II B-7
B.4 VAX-11/725 . . . . B-7
B.5 VAX-11/730 . . . . B-7
B.6 VAX-11/750 . . . . B-8
B.7 VAX-11/780 . . . . B-8
B.8 VAX-11/782 . . . . B-8
B.9 VAX-11/785 . . . . B-8
B.10 VAX 8200 . . . . B-9
B.11 VAX 8300 . . . . B-9

B.12 VAX 8500 . . . . B-9
B.13 VAX 8600 . . . . B-9
B.14 VAX 8650 . . . . B-9
B.15 VAX 8700 . . . . B-10
B.16 VAX 8800 . . . . B-10
B.17 VVAX B-10

APPENDIX C ARCHITECTURE MANAGEMENT

C.1 RESPONSIBILITIES C-1
C.2 SPECIFICATION DISTRIBUTION C-3
C.3 CONFORMANCE AND WAIVERS C-4
C.4 ECO PROCESS C-5
C.5 SYSTEMS-ARCHITECTURE-GROUP SPECIFICATIONS C-8

INDEX

FIGURES

1-1 Data Types 1-5
1-2 Floating-Point Data Types 1-8
1-3 Variable-Length Bit Fields 1-11
1-4 Absolute Queues 1-13
1-5 Self-Relative Queues 1-15
1-6 Character-String Data Type 1-19
1-7 Decimal Strings 1-21
1-8 Processor State 1-25
2-1 Opcodes and Operand Specifiers 2-2
2-2 Addressing-Mode Specifiers 2-5
2-3 Literal Address Mode 2-11
2-4 Interpretation of a Literal 2-12
2-5 Indexed Specifier and Extension 2-16
2-6 Branch-Mode Specifiers 2-16
3-1 Procedure Entry Mask 3-72
3-2 Procedure-Call Stack Frame 3-72
3-3 Result Registers After POLYF 3-140
3-4 Result Registers After POLYD 3-140
3-5 Result Registers After POLYG 3-140
3-6 Result Registers After POLYH 3-141
3-7 Character-String-Instruction Control Block . . . 3-152
3-8 Decimal-String-Instruction Control Block 3-184
3-9 EDITPC Control Block 3-220
4-1 Virtual-Address Space 4-3
4-2 Virtual-Address Format 4-3
4-3 Address Space Number Register (ASN) 4-5
4-4 Map Enable Register (MAPEN) 4-10
4-5 Page Table Entry (25-Bit Page Frame Number) . . 4-10
4-6 Page Table Entry (21-Bit Page Frame Number) . . 4-11
4-7 System Virtual-Address Format 4-15
4-8 System Page-Table Mapping Registers 4-18
4-9 System Virtual-Address Translation (25-Bit PFN) 4-19
4-10 System Virtual-Address Translation (21-Bit PFN) 4-20
4-11 Process Virtual-Address Format 4-23
4-12 Process-Space Mapping Registers 4-23
4-13 Process Virtual-Address Translation 4-24
4-14 TB Invalidate (based on) Address Space Number (TBIASN) 4-29
4-15 Memory-Management-Fault Stack Frame 4-31
5-1 Software-Interrupt Summary Register 5-8
5-2 Software-Interrupt Request Register 5-8
5-3 Interrupt-Priority-Level Register 5-8
5-4 Arithmetic-Exception Stack Frame 5-10
5-5 CHMx Stack Frame 5-20
5-6 An Example Machine-Check Stack Frame 5-20
5-7 System Control-Block Base 5-20
6-1 Process Control Block (PCB) 6-3
6-2 Process Control-Block Base Register (PCBB) 6-4
6-3 Performance-Monitor-Enable Register (PME) 6-4
6-4 AST-Level Register (ASTLVL) 6-4
7-1 PTEs For Use By I/O Devices 7-21
8-1 CPU Identification Register (CPUID) 8-17
8-2 System Identification Register (SID) 8-21
8-3 System-Type Register (SYS_TYPE) 8-21
8-4 Time of Year (TODR) 8-24
8-5 Clock Registers 8-24

9-1 Compatibility-Mode Operand Specifiers 9-5
9-2 Compatibility-Mode Instruction Formats 9-8
10-1 Restart Parameter Block (RPB) 10-6
10-2 Console Terminal Registers 10-34
11-1 rtVAX Process-Space Mapping Registers 11-7
11-2 rtVAX Process-Space Address Translation 11-8
11-3 Instruction-Emulation Trap Frame 11-15
12-1 Real VAXes and Virtual VAXes 12-3
12-2 Virtual-Machine PSL Register (VMPSL) 12-4
12-3 VM-Emulation-Trap Stack Frame 12-14
12-4 Use of Access Modes on Virtual VAX 12-17
13-1 Vector Register 13-6
13-2 Vector Length Register (VLR) 13-6
13-3 Vector Mask Register (VMR) 13-6
13-4 Vector Count Register (VCR) 13-6
13-5 Vector Processor Status Register (VPSR) 13-6
13-6 Vector Arithmetic Exception Register (VAER) . . 13-10
13-7 Vector Memory Activity Check Register (VMAC) . . 13-10
13-8 Vector Translation Buffer Invalidate All Register (VTBIA) 13-10
13-9 Vector State Address Register (VSAR) 13-11
13-10 Vector Control Word Operand (cntrl) 13-18
13-11 Vector Control Word Format 13-21
13-12 Encoding of the Reserved Operand 13-44
14-1 Memory Management Fault Stack Frame 14-18
B-1 Conceptual Configuration of a VVAX System . . . B-11
B-2 MicroVAX I Physical Address Space B-14
B-3 MicroVAX I System Identification Register (SID) B-15
B-4 MicroVAX I Machine-Check Stack Frame B-16
B-5 MicroVAX II Physical Address Space B-17
B-6 MicroVAX II System Identification Register (SID) B-19
B-7 MicroVAX II Machine-Check Stack Frame B-19
B-8 VAX-11/730 Physical Address Space B-21
B-9 VAX-11/730 System Identification Register (SID) B-23
B-10 VAX-11/730 Machine-Check Stack Frame B-24
B-11 VAX-11/750 Physical Address Space B-25
B-12 VAX-11/750 System Identification Register (SID) B-27
B-13 VAX-11/750 Machine-Check Stack Frame B-28
B-14 VAX-11/780 Physical Address Space B-29
B-15 VAX-11/780 System Identification Register (SID) B-31
B-16 VAX-11/780 Machine-Check Stack Frame B-32
B-17 VAX-11/8200 Physical Address Space B-33
B-18 VAX 8200 System Identification Register (SID) . B-35
B-19 VAX 8200 Machine-Check Stack Frame B-36
B-20 VAX-11/8600 Physical Address Space B-37
B-21 VAX 8600 System Identification Register (SID) . B-40
B-22 VAX 8600 Machine-Check Stack Frame B-41
B-23 VAX-11/8800 Physical Address Space B-43
B-24 VAX 8800 System Identification Register (SID) . B-45
B-25 VAX 8800 Machine-Check Stack Frame B-46
B-26 VVAX Physical Address Space B-47
B-27 VVAX System Identification Register (SID) . . . B-48
B-28 VVAX MEMSIZE Register B-49
B-29 VVAX KCALL Register B-49
B-30 VVAX Machine-Check Stack Frame B-49

TABLES

1-1 Sign and Digits in Decimal Strings 1-20
1-2 Fields of the Processor Status Longword 1-26
2-1 Floating-Point Values Representable as Literals 2-11
2-2 Summary of General Register Addressing 2-17
3-1 Common CRC Polynomials 3-180
3-2 CRC Shift Amounts 3-180
3-3 EDIT Pattern Operators 3-220
3-4 EDIT Pattern-Operator Encoding 3-221
4-1 Fields of the PTE (25-Bit Page Frame Number) . . 4-10
4-2 Fields of the PTE (21-Bit Page Frame Number) . . 4-11
4-3 PTE Protection Codes 4-15
4-4 Fields of the Memory-Management-Fault Parameter 4-32
5-1 An Example of Interrupt Processing 5-8
5-2 Arithmetic-Exception Type Codes 5-10
5-3 Compatibility-Mode Exception Type Codes 5-10
5-4 State Changes During Tracing 5-17
5-5 System-Control-Block Vectors 5-25
5-6 Indication of Current Stack Pointer 5-26
5-7 Processor State Transitions 5-36
6-1 Contents of the Process Control Block 6-4
7-1 Resources that when changed can modify the Instruction Stream 7-13
7-2 PTE Types For Use By I/O 7-21
8-1 Required Internal Processor Registers 8-7
8-2 All Internal Processor Registers 8-9
8-3 SID<31:24> and SYS_TYPE<31:24> Assignments . . . 8-19
8-4 Bit Assignments in the LICENSE_ID Subfield . . . 8-21
8-5 Fields of the ICCS Register 8-25
9-1 Compatibility-Mode Instructions 9-7
9-2 Compatibility-Mode Trap Instructions 9-8
9-3 Compatibility-Mode Reserved Instructions 9-8
10-1 Major System State Transitions 10-6
10-2 Effects of Processor Initialization 10-8
10-3 Additional Effects of Power-Up Initialization . 10-8
10-4 Fields of the RXCS Register 10-35
10-5 Fields of the RXDB Register 10-35
10-6 Fields of the TXCS Register 10-36
10-7 Fields of the TXDB Register 10-36
12-1 Fields of the Virtual Machine's PSL 12-4
12-2 Contents of the VM-Emulation-Trap Stack Frame . 12-14
13-1 Description of the Vector Processor Status Register 13-7
13-2 Possible VPSR<3:0> Settings For MTPR 13-9
13-3 State of the Vector Processor 13-9
13-4 VAER Exception Condition Summary Word Encoding . 13-10
13-5 IPR Assignments 13-11
13-6 Description of the Vector Control Word Operand . 13-18
13-7 Encoding of the Exception Condition Type (ETYPE) 13-44
13-8 Assembler Notation for Vector Instructions . . . 13-66
14-1 Dependences for Vector Operate Instructions . . 14-11
14-2 Dependences for Vector Load and Gather Instructions 14-12
14-3 Dependences for Vector Store and Scatter Instructions 14-12
14-4 Dependences for Vector Compare Instructions . . 14-13
14-5 Dependences for Vector MERGE Instructions . . . 14-13

14-6 Dependences for IOTA Instruction 14-13
14-7 Dependences for MFVP Instructions 14-14
14-8 Miscellaneous Dependences 14-14
14-9 When Scalar/Vector Memory Synchronization (M) Or VSYNC (V) 14-37
A-1 Single Byte Opcodes A-1
A-2 Two-Byte Opcodes A-5
A-3 Vector Opcodes A-8
A-4 Vector Opcode Groupings By Format A-10
B-1 MicroVAX I Implementation-Dependent SCB Vectors B-14
B-2 MicroVAX I Halt Codes B-14
B-3 MicroVAX I Implementation-Dependent IPRs B-15
B-4 MicroVAX I Machine-Check Type Codes B-16
B-5 MicroVAX II Implementation-Dependent SCB Vectors B-18
B-6 MicroVAX II Implementation-Dependent IPRs . . . B-18
B-7 MicroVAX II Machine-Check Type Codes B-19
B-8 MicroVAX II Halt Codes B-20
B-9 VAX-11/730 Implementation-Dependent SCB Vectors B-22
B-10 VAX-11/730 Halt Codes B-22
B-11 VAX-11/730 Implementation-Dependent IPRs B-23
B-12 VAX-11/730 Machine-Check Type Codes B-24
B-13 VAX-11/750 Implementation-Dependent SCB Vectors B-26
B-14 VAX-11/750 Halt Codes B-26
B-15 VAX-11/750 Implementation-Dependent IPRs B-27
B-16 VAX-11/750 Machine-Check Error Summary Register B-28
B-17 VAX-11/780 Implementation-Dependent SCB Vectors B-30
B-18 VAX-11/780 Halt Codes B-30
B-19 VAX-11/780 Implementation-Dependent IPRs B-31
B-20 VAX-11/780 Machine-Check Error Summary Parameter B-32
B-21 VAX-11/8200 Implementation-Dependent SCB Vectors B-34
B-22 VAX 8200 Implementation-Dependent IPRs B-35
B-23 VAX-11/8600 Implementation-Dependent SCB Vectors B-38
B-24 VAX 8600 Halt Codes B-39
B-25 VAX 8600 Implementation-Dependent IPRs B-40
B-26 VAX 8600 Machine-Check Stack Frame Contents . . B-42
B-27 VAX-11/8800 Implementation-Dependent SCB Vectors B-44
B-28 VAX 8800 BI Vectors (VMS V4.4) B-44
B-29 VAX 8800 Implementation-Dependent IPRs B-45
B-30 VAX 8800 Machine-Check Stack Frame Contents . . B-46
B-31 VVAX Implementation-Dependent SCB Vectors . . . B-47
B-32 VVAX Implementation-Dependent IPRs B-48
B-33 Fields of the VVAX SID Register B-48
B-34 VVAX Machine-Check Type Code Parameter B-49
B-35 VVAX Implementation-Dependent Halt Codes B-50

PREFACE

VAX is a family of upward-compatible computer systems. It is a natural outgrowth of and is strongly compatible with the PDP-11 family.

This document is the definition of the VAX architecture. It provides a complete description of the VAX central-processor hardware as seen by machine-language programs, and applies to all software written for VAX processors, all VAX central-processor hardware, and all "closely coupled" VAX hardware peripherals. The definition of the central processor describes 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 use 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. Moreover, the manual is completely self-contained -- no prior knowledge of the assembler is required.

Readers interested in just a summary of the family should refer to the VAX Technical Summary.

To conform to this standard, a processor must correctly run these tests:

1. The VAX architecture-conformance tests, AXE and MAX, described below

2. The VAX Cluster Exerciser diagnostics

3. VMS

4. UETP, the VMS User-Environment Test Package

5. The quality-assurance packages for layered software

The AXE portion of the VAX architecture conformance test divides testing into five parts: testing the nineteen unprivileged instruction groups, testing PDP-11 compatibility mode, testing the eleven privileged instructions, testing the eleven implicit stack instructions, and testing PSL<FPD> handling.

The nineteen unprivileged instruction groups are:

Bit Field
Branch
Character String
D_floating
Decimal String
Emulated String
F_floating
G_floating
H_floating
Integer
Miscellaneous
No Operand Specifier
OPCDEC
POLYD
POLYF
POLYG
POLYH
Privileged Instructions in User Mode
Queue

The eleven privileged instructions are CHMK, CHME, CHMS, CHMU, LDPCTX, MFPR, MTPR, PROBER, PROBEW, SVPCTX, and REI.

The eleven implicit stack instructions are CALLG, CALLS, POPR, PUSHAB, PUSHAW, PUSHAL, PUSHAD, PUSHAH, PUSHL, PUSHR, and RET.

PSL<FPD> testing involves interrupting complex instructions during their execution, replacing their saved state with invalid values, and resuming their execution. The testing is intended to ensure that the machine cannot be crashed or hung from user mode.

The MAX portion of VAX architecture conformance testing consists of test runs with Multi-instruction AXE, which extends testing to include such areas as inter- and intra-instruction resource dependencies, overlapping operands, and vector instructions. Because MAX is currently under development, the minimum requirement is subject to change. Therefore, the Architecture Verification Group (the AXE Group) should be contacted for current test requirements.

To comply with the AXE portion of the VAX architecture conformance requirement, a processor must execute a minimum of 10,000,000 error-free AXE test cases in each unprivileged instruction group, 5,000,000 error-free test cases in each PDP-11 compatibility mode group (if the processor implements compatibility mode), 5,000 error-free test cases of each privileged-mode and stack instruction, and 10,000,000 cases of PSL<FPD> testing.

To ensure conformance, AXE and MAX testing for DEC STD 032 must be done using data files provided by the Architecture Verification Group. The processor must be fully tested in every revision level of each processor configuration that will be shipped, such as with or without floating-point hardware or vector processor.

Final testing of the prototype must be done by either the AXE group or the development group under the supervision of the AXE group. If the development group conducts the verification, they must certify that the verification criteria were met by submitting to the AXE group detailed logs of the testing process and descriptions of the revision levels tested. The testing of the prototype may take place at the AXE group site, at the development site, or at both.

AXE and MAX testing must continue beyond shipment of the initial release. It is the responsibility of the development group to ensure that the appropriate support group has access to a system for the life of the product. The support group is responsible for ensuring that at least the minimum testing is done on each new release of microcode or hardware before it ships.

AXE and MAX are also valuable tools for debugging designs that are under development. In order to ensure successful final verification, the development group should begin testing with AXE and MAX as early as possible, including during the simulation phase. The Architecture Verification Group will assist the development group in setting-up and running AXE and MAX and in analyzing any bugs that are discovered.

Details of AXE and MAX functions are included in the AXE and MAX User's Guides. Additional information and assistance are available from the Architecture Verification Group.

The Architecture Group also maintains lists of the bugs in announced VAX processors. The lists can be found in the directory:

```
EAGLE1::SYS$PUBLIC:[VAXBUG]
```

To get access to this directory, contact the Architecture Group at:

```
EAGLE1::SRM
```

Change History:

Revision J. Larry Camilli and Rich Brunner, December 1989.
- o Revise the description of AXE and add description of MAX.
- o AXE and MAX testing must continue beyond shipment of the initial release. It is the responsibility of the development group to ensure that the appropriate support group has access to a system for the life of the product. The support group is responsible for ensuring that at least the minimum testing is done on each new release of microcode or hardware before it ships.

Revision H. Tim Leonard, May 1987.
- o Revise the description of AXE.
- o Create Appendix C to contain the Architecture-Management Process Document.

Revision F. Al Thomas, November 1986.
- o Include Revision 4 of the Architecture Management Process Document.

Revision D1. Al Thomas, September 1986.
- o Rewrite the Preface to account for the Architecture Management Process Document
- o Include the Architecture Management Process Document.

Revision D. Tim Leonard, March 1985.
- o Correct the VAXB change process to conform to practice.
- o Merge much of the header for the DEC Standard with the Preface.
- o Rewrite the definition of who's on VAXB.

Revision A. Tom Eggers, 10 July 1980.
- o Make the SRM a DEC Standard.

CHAPTER 1

BASIC ARCHITECTURE

The VAX architecture represents a significant extension of the PDP-11 family architecture. It shares byte addressing with the PDP-11, 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 allows straightforward manual conversion of existing PDP-11 programs to the VAX system. Existing user-mode PDP-11 programs which do not need the extended features of VAX can run unchanged in the PDP-11 compatibility mode provided in the VAX architecture.

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

A number of specific goals are achieved in the VAX design:

- VAX architecture has maximal compatibility with the PDP-11 consistent with a significant extension of the virtual address space and a significant functional enhancement.

- High bit efficiency is achieved by a wide range of data types and new addressing modes.

- The systematic, elegant instruction set with orthogonality of operators, data types, and addressing modes can be exploited easily, particularly by high-level language processors.

- The VAX system is extensible. 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.

- The architecture is suitable in terms of price and performance over a wide range of computer system implementations sold by Digital Equipment Corporation.

Certain terminology and conventions are used throughout this book. All numbers unless otherwise indicated are decimal. Where there is ambiguity, the radix is explicitly stated, as in 48 (hex), or 1001000 (binary).

With the addition of vector processing, the VAX architecture can be classified into two parts: a vector part which includes the instructions, registers, and execution model for vector processing; and a scalar part which includes the remainder of the architecture. Where confusion may be possible, this book uses the term "scalar" to describe objects belonging to the scalar part of the architecture -- as in scalar instructions and scalar processor. Similarly, the term "vector" is used to describe objects belonging to the vector part of the architecture -- as in vector registers and vector instructions. With the exception of chapters 13 and 14, instructions, exceptions, registers, and other objects described in the rest of this book refer to the scalar part of the architecture unless otherwise stated.

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 (reach an unhalted state from which there is no transition to a normal state in which the processor executes instructions). Note the distinction between result and operation: non-privileged software cannot invoke UNDEFINED operations. When the operation of the VAX scalar processor becomes UNDEFINED, so does the operation of its associated vector processor. The converse is not true; when the operation of the vector processor becomes UNDEFINED, the operation of the scalar processor need not become UNDEFINED.

Ranges are specified in English and are inclusive. For example, 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. For example, bits <7:3> specifies an extent of bits including bits 7, 6, 5, 4, and 3.

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 if that field is accessible to non-privileged software. (See Chapter 5, Interrupts and Exceptions, for a description of faults and aborts.) MBZ fields that are accessible only to privileged software (kernel mode) may not be checked for non-zero value by some or all VAX implementations. Non-zero values in MBZ fields accessible only to privileged software may produce UNDEFINED operation.

Fields specified as SBZ (Should Be Zero) can produce UNPREDICTABLE results if filled by software with Non-zero values. Unassigned values of fields are reserved for future use. In many cases, some values are indicated as reserved for the customer, that is, the equipment owner. Only these values should be used for non-standard applications. The values indicated as reserved for DIGITAL and all MBZ fields are to be used only to extend the standard architecture in the future.

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

NOTE

> \At certain points in the manual, comments for internal use only are included between a pair of backslashes. The comments typically explain decisions made in the design of the architecture, or give implementation hints or warnings.\

1.1 ADDRESSING

The basic addressable unit in the VAX architecture 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 4.

1.2 DATA TYPES

Following are descriptions of the VAX architecture data types.

1.2.1 Byte

A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from the right <0> through <7>, as shown in Figure 1-1a. A byte is specified by its address A. When interpreted arithmetically, a byte is a two's complement integer with bits of increasing significance from <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 instructions also provide direct support for the interpretation of a byte as an unsigned integer with bits of increasing significance from <0> through <7>. The value of the unsigned integer is in the range 0 through 255.

1.2.2 Word

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

1-1b. 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 from <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 instructions also provide direct support for the interpretation of a word as an unsigned integer with bits of increasing significance from <0> through <15>. The value of the unsigned integer is in the range 0 through 65,535.

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

a. Byte

```
                                15                             0
                                +-------------------------------+
                                |                               | :A
                                +-------------------------------+
```

b. Word

```
31                                                             0
+---------------------------------------------------------------+
|                                                               | :A
+---------------------------------------------------------------+
```

c. Longword

```
31                                                             0
+---------------------------------------------------------------+
|                                                               | :A
+---------------------------------------------------------------+
|                                                               | :A+4
+---------------------------------------------------------------+
63                                                             32
```

d. Quadword

```
31                                                             0
+---------------------------------------------------------------+
|                                                               | :A
+---------------------------------------------------------------+
|                                                               | :A+4
+---------------------------------------------------------------+
|                                                               | :A+8
+---------------------------------------------------------------+
|                                                               | :A+12
+---------------------------------------------------------------+
127                                                            96
```

e. Octaword

Figure 1-1 Data Types

1.2.3 Longword

A longword is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right <0> through <31>, as shown in Figure 1-1c. 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 from <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 instructions also provide direct support for the interpretation of a longword as an unsigned integer with bits of increasing significance from <0> through <31>. The value of the unsigned integer is in the range 0 through 4,294,967,295.

1.2.4 Quadword

A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right <0> through <63>, as shown in Figure 1-1d. 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 from <0> through <62> and bit <63>, the sign bit. The value of the integer is in the range -2**63 to 2**63-1. Only a subset of the full complement of operators is provided for quadword.

1.2.5 Octaword

(The octaword data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) An octaword is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right <0> through <127>, as shown in Figure 1-1e. An 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 from <0> through <126> and bit <127>, the sign bit. The value of the integer is in the range -2**127 to 2**127-1. Only a subset of the full complement of operators is provided for octaword.

1.2.6 F_floating

An F_floating datum is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right <0> through <31>, as shown in Figure 1-2a. An F_floating datum is specified by its address A, the address of the byte containing bit <0>. The form of an F_floating datum is sign magnitude with bit <15>, the sign bit; bits <14:7>, an excess 128 binary exponent; and bits <6:0> and <31:16>, a normalized 24-bit fraction with the redundant most-significant fraction bit not represented. Within the fraction, bits of increasing significance go from <16> through <31> and <0> through <6>. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0 together with a sign bit of 0 is taken to indicate that the F_floating datum has a value of 0. Exponent values of 1 through 255 indicate true binary exponents of -127 through +127. An exponent value of 0 together with a sign bit of 1 is taken as reserved. Floating-point instructions processing a reserved operand take a reserved operand fault. (See Chapter 5, Interrupts and Exceptions, for a description of faults.) The value of an F_floating datum is in the approximate range .29*10**-38 through 1.7*10**38. The precision of an F_floating datum is approximately one part in 2**23, typically 7 decimal digits.

1.2.7 D_floating

A D_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right <0> through <63>, as shown in Figure 1-2b. 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 an F_floating datum except for an additional 32 low-significance fraction bits. Within the fraction, bits of increasing significance are from <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 for F_floating. The precision of a D_floating datum is approximately one part in 2**55, typically 16 decimal digits.

```
31                            16 15 14           7 6           0
+-------------------------------+-+---------------+-------------+
|            fraction           |S|    exponent   |  fraction   | :A
+-------------------------------+-+---------------+-------------+
```

a. F_floating Data Type (Single Precision)

```
31                            16 15 14           7 6           0
+-------------------------------+-+---------------+-------------+
|            fraction           |S|    exponent   |  fraction   | :A
+-------------------------------+-+---------------+-------------+
|            fraction           |           fraction            | :A+4
+-------------------------------+-------------------------------+
63                                                             32
```

b. D_floating Data Type (Double Precision)

```
31                            16 15 14                 4 3     0
+-------------------------------+-+---------------------+-------+
|            fraction           |S|      exponent       |  fra  | :A
+-------------------------------+-+---------------------+-------+
|            fraction           |           fraction            | :A+4
+-------------------------------+-------------------------------+
63                                                             32
```

c. G_floating Data Type (Extended-Range Double Precision)

```
31                            16 15 14                         0
+-------------------------------+-+-----------------------------+
|            fraction           |S|         exponent            | :A
+-------------------------------+-+-----------------------------+
|            fraction           |           fraction            | :A+4
+-------------------------------+-------------------------------+
|            fraction           |           fraction            | :A+8
+-------------------------------+-------------------------------+
|            fraction           |           fraction            | :A+12
+-------------------------------+-------------------------------+
127                                                            96
```

d. H_floating Data Type (Extended-Range Quadruple Precision)

Figure 1-2 Floating-Point Data Types

1.2.8 G_floating

A G_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right <0> through <63>, as shown in Figure 1-2c. 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 are from <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 5, Interrupts and Exceptions, for a description of faults.) 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, typically 15 decimal digits.

1.2.9 H_floating

(The H_floating data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) An H_floating datum is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right <0> through <127>, as shown in Figure 1-2d. An H_floating datum is specified by its address A which is the address of the byte containing bit <0>. The form of an 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 are from <112> through <127>, <96> through <111>, <80> through <95>, <64> through <79>, <48> through <63>, <32> through <47>, and <16> through <31>. The 15-bit exponent field encodes the values 0 through 32767. An exponent value of 0 together with a sign bit of 0 is taken to indicate that the H_floating datum has a value of 0. Exponent values of 1 through 32767 indicate true binary exponents of -16383 through +16383. An exponent value of 0 together with a sign bit of 1 is taken as reserved. Floating-point instructions processing a reserved operand take a reserved operand fault. (See Chapter 5, Interrupts and Exceptions, for a description of faults.) The value of an H_floating datum is in the approximate range .84*10**-4932 through .59*10**4932. The precision of an H_floating datum is approximately one part in 2**112, typically 33 decimal digits.

1.2.10 Variable-Length Bit Field

A variable-length bit field is 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable-length bit field is specified by three 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, as shown in Figure 1-3a.

For bit fields 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, as shown in Figure 1-3b. 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 field instructions provide direct support for the interpretation of a field as a signed or unsigned integer. When interpreted as a signed integer, a field contains a two's complement integer with bits of increasing significance from 0 through S-2; bit S-1 is the sign bit. When interpreted as an unsigned integer, bits of increasing significance are from 0 to S-1. A field of size 0 has a value of 0.

A variable-length bit field may be contained in 1 to 5 bytes. Instructions that read or write fields in memory reference only the minimum number of aligned longwords necessary to contain the field. (For more detail about how fields are referenced in memory, see section 7.3, Memory References.)

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-length bit field may be contained in two registers if the sum of position and size exceeds 32, as shown in Figure 1-3c.

See the descriptions of the bit-field instructions in Chapter 3 for further details on the specification of variable-length bit fields.

```
             P+S P+S-1                P P-1                    0
+---------------+----------------------+------------------------+
|               |//////////////////////|                        | :A
+---------------+----------------------+------------------------+
                 S-1                  0
```

a. Variable-Length Bit Field Data Type In Memory

```
31                                                     3 2     0
+-------------------------------------------------------+-------+
|                    byte offset                        | bwb   |
+-------------------------------------------------------+-------+
```

b. Bit Field Position

```
31       P P-1                                                 0
+---------+-----------------------------------------------------+
|/////////|                                                     | Rn
+---------------------------------------------------------------+
|                                                   |///////////| R[n+1]
+---------------------------------------------------+-----------+
                                                P+S P+S-1
```

c. Variable-Length Bit Field Data Type across a Register Boundary

Figure 1-3 Variable-Length Bit Fields

1.2.11 Absolute Queues

A queue is a circular, doubly linked list. A queue entry is specified by its address. Each queue entry is linked to the next with a pair of longwords. A queue is classified by the type of link it uses. Absolute queues use absolute addresses as links.

The first (lowest addressed) longword is the forward link; it specifies the address of the succeeding queue entry. The second (highest addressed) longword is the backward link; it specifies 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.

An empty queue is specified by its header at address H, as shown in Figure 1-4a. If an entry at address B is inserted into an empty queue (at either the head or tail), the queue shown in Figure 1-4b results. Figures 1-4c, d, and e, respectively, illustrate the results of subsequent insertion of an entry at address A at the head, insertion of an entry at address C at the tail, and removal of the entry at address B.

1.2.12 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; it is a displacement of the succeeding queue entry from the present entry. The second longword (highest addressed) is the backward link; it is the displacement of the preceding queue entry from the present entry. A queue is specified by a queue header, which also consists of two longword links.

An empty queue is specified by its header at address H. Since the queue is empty, the self-relative links are zero, as shown in Figure 1-5a.

Figures 1-5b, c, and d, respectively, illustrate the results of subsequent insertion of an entry at address B at the head, insertion of an entry at address A at the tail, and insertion of an entry at address C at the tail.

Figures 1-5d, c and b (in that order) illustrate the effect of removal at the tail and removal at the head.

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

a. An Empty Absolute Queue

```
+---------------------------------------------------------------+
|                              B                                | :H
+---------------------------------------------------------------+
|                              B                                | :H+4
+---------------------------------------------------------------+


+---------------------------------------------------------------+
|                              H                                | :B
+---------------------------------------------------------------+
|                              H                                | :B+4
+---------------------------------------------------------------+
```

b. An Absolute Queue with One Entry

```
+---------------------------------------------------------------+
|                              A                                | :H
+---------------------------------------------------------------+
|                              B                                | :H+4
+---------------------------------------------------------------+


+---------------------------------------------------------------+
|                              B                                | :A
+---------------------------------------------------------------+
|                              H                                | :A+4
+---------------------------------------------------------------+


+---------------------------------------------------------------+
|                              H                                | :B
+---------------------------------------------------------------+
|                              A                                | :B+4
+---------------------------------------------------------------+
```

c. An Absolute Queue with Two Entries

Figure 1-4 Absolute Queues

```
+---------------------------------------------------------------+
|                               A                               | :H
+---------------------------------------------------------------+
|                               C                               | :H+4
+---------------------------------------------------------------+


+---------------------------------------------------------------+
|                               B                               | :A
+---------------------------------------------------------------+
|                               H                               | :A+4
+---------------------------------------------------------------+


+---------------------------------------------------------------+
|                               C                               | :B
+---------------------------------------------------------------+
|                               A                               | :B+4
+---------------------------------------------------------------+


+---------------------------------------------------------------+
|                               H                               | :C
+---------------------------------------------------------------+
|                               B                               | :C+4
+---------------------------------------------------------------+
```

d. An Absolute Queue With Three Entries

```
+---------------------------------------------------------------+
|                               A                               | :H
+---------------------------------------------------------------+
|                               C                               | :H+4
+---------------------------------------------------------------+


+---------------------------------------------------------------+
|                               C                               | :A
+---------------------------------------------------------------+
|                               H                               | :A+4
+---------------------------------------------------------------+


+---------------------------------------------------------------+
|                               H                               | :C
+---------------------------------------------------------------+
|                               A                               | :C+4
+---------------------------------------------------------------+
```

e. An Absolute Queue with Three Entries After Removing the Second Entry

Figure 1-4 Absolute Queues (continued)

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

a. An Empty Self-Relative Queue

```
+-----------------------------------------------------------------+
|                             B - H                               | :H
+-----------------------------------------------------------------+
|                             B - H                               | :H+4
+-----------------------------------------------------------------+


+-----------------------------------------------------------------+
|                             H - B                               | :B
+-----------------------------------------------------------------+
|                             H - B                               | :B+4
+-----------------------------------------------------------------+
```

b. A Self-Relative Queue with One Entry

```
+-----------------------------------------------------------------+
|                             A - H                               | :H
+-----------------------------------------------------------------+
|                             B - H                               | :H+4
+-----------------------------------------------------------------+


+-----------------------------------------------------------------+
|                             B - A                               | :A
+-----------------------------------------------------------------+
|                             H - A                               | :A+4
+-----------------------------------------------------------------+


+-----------------------------------------------------------------+
|                             H - B                               | :B
+-----------------------------------------------------------------+
|                             A - B                               | :B+4
+-----------------------------------------------------------------+
```

c. A Self-Relative Queue with Two Entries

Figure 1-5 Self-Relative Queues

```
+-------------------------------------------------------------+
|                            A - H                            | :H
+-------------------------------------------------------------+
|                            C - H                            | :H+4
+-------------------------------------------------------------+

+-------------------------------------------------------------+
|                            B - A                            | :A
+-------------------------------------------------------------+
|                            H - A                            | :A+4
+-------------------------------------------------------------+

+-------------------------------------------------------------+
|                            C - B                            | :B
+-------------------------------------------------------------+
|                            A - B                            | :B+4
+-------------------------------------------------------------+

+-------------------------------------------------------------+
|                            H - C                            | :C
+-------------------------------------------------------------+
|                            B - C                            | :C+4
+-------------------------------------------------------------+
```

d. A Self-Relative Queue with Three Entries

Figure 1-5 Self-Relative Queues (continued)

1.2.13 Character String

A character string is a contiguous sequence of bytes in memory. A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. The address of a string specifies the first character of a string. See Figure 1-6.

The length L of a string is in the range 0 through 65,535.

1.2.14 Trailing Numeric String

(The trailing-numeric-string data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) A trailing numeric string is a contiguous sequence of bytes in memory. The string is specified by two 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 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. There are, however, three preferred encodings used by DIGITAL software: (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 many compiler 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 encoding of sign and digits in trailing numeric strings is shown in Table 1-1.

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

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. Figures 1-7a and 1-7b illustrate the representation of trailing numeric strings in zoned and overpunch formats.

1.2.15 Leading-Separate Numeric String

(The leading-separate-numeric-string data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) A leading-separate numeric string is a contiguous sequence of bytes in memory. A leading-separate numeric string is specified by two 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 leading-separate numeric string is stored in a separate byte. Each subsequent byte contains an ASCII digit character. The signs and digits of leading-separate numeric strings are shown in Table 1-1.

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

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. Figure 1-7c illustrates the representation of leading-separate numeric strings.

1.2.16 Packed Decimal String

(The packed-decimal-string data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) A packed decimal string is a contiguous sequence of bytes in memory. A packed decimal string is specified by two 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 two, 4-bit fields that must contain decimal digits, with the exception of the low nibble (bits <3:0>) of the last (highest addressed) byte that must contain a sign.

The preferred sign representation is 0C (hex) for positive and 0D (hex) for negative, as shown in Table 1-1.

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. Figure 1-7d illustrates the representation of packed decimal strings.

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

    Character String of Length L

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

    Character String "XYZ"

Figure 1-6 Character-String Data Type

Table 1-1: Sign and Digits in Decimal Strings

	Zoned Trailing Numeric		Overpunch Trailing Numeric		Leading Separate Numeric		Packed Decimal
	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex
Sign							
positive					2B	+	C
positive *					20	<blank>	A E F
negative					2D	-	D
negative *							B
Digit							
0	30	0	30	0	30	0	0
1	31	1	31	1	31	1	1
2	32	2	32	2	32	2	2
3	33	3	33	3	33	3	3
4	34	4	34	4	34	4	4
5	35	5	35	5	35	5	5
6	36	6	36	6	36	6	6
7	37	7	37	7	37	7	7
8	38	8	38	8	38	8	8
9	39	9	39	9	39	9	9
Combined sign and digit							
+0	30	0	7B	{			
+1	31	1	41	A			
+2	32	2	42	B			
+3	33	3	43	C			
+4	34	4	44	D			
+5	35	5	45	E			
+6	36	6	46	F			
+7	37	7	47	G			
+8	38	8	48	H			
+9	39	9	49	I			
-0	70	p	7D	}			
-1	71	q	4A	J			
-2	72	r	4B	K			
-3	73	s	4C	L			
-4	74	t	4D	M			
-5	75	u	4E	N			
-6	76	v	4F	O			
-7	77	w	50	P			
-8	78	x	51	Q			
-9	79	y	52	R			

* These alternative representations of the sign are permitted. VAX instructions always produce the preferred representation, which is shown first.

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

a. "+123" and "-12" as Zoned-Format Trailing-Numeric Strings

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

b. "+123" and "-12" as Overpunch-Format Trailing-Numeric Strings

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

c. "+123" and "-12" as Leading-Separate Numeric Strings

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

d. "+123" and "-12" as Packed Decimal Strings

Figure 1-7 Decimal Strings

1.3 PROCESSOR STATE

The processor state consists of that portion of a process's state that, while the process is executing, is stored in processor registers rather than memory.

- Sixteen 32-bit general-purpose registers denoted Rn or R[n], where n is in the range 0 through 15

- A 32-bit processor status longword (PSL)

- Privileged internal processor registers (IPR)

1.3.1 General-Purpose Registers

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 is termed an index register. The bits of a register are numbered from the right <0> through <31>, as shown in Figure 1-8a.

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

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

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

- R13 is the current frame pointer (FP). The VAX procedure call convention builds a data structure on the stack called a stack frame. FP contains the address of the base of this data structure.

- R12 is the argument pointer (AP). The VAX 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. As will be seen in Chapter 2, however, PC cannot be used as an accumulator, temporary, or index register.

When a datum of type byte, word, longword, or F_floating is stored in a register, the bit numbering in the register corresponds to the numbering in memory. Hence a byte is stored in register bits <7:0>, a word in register bits <15:0>, and a 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 two adjacent registers R[n] and R[n+1]. Because of restrictions on the uses of PC (see Chapter 2), wraparound from PC to R0 and from SP to PC produces UNPREDICTABLE results. 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 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 2), wraparound from PC to R0 and from AP, FP, and SP to PC produces UNPREDICTABLE results. 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].

A variable-length bit field may be specified in the registers with the restriction that the starting bit position P must be in the range 0 through 31. See Figure 1-3c. 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 bits <63:32> in register R[n+1].

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

1.3.2 Processor Status Longword

The processor status longword (PSL) is a longword consisting of a word of privileged processor status concatenated with the processor status word (PSW), as shown in Figure 1-8c and described in Table 1-2. The processor status word (PSW) contains the condition codes that give information on the results produced by previous VAX scalar instructions and the exception-enable bits which control the processor action on certain VAX scalar exception conditions.

(See Chapter 5, Interrupts and Exceptions, for more about exceptions.) The condition codes are UNPREDICTABLE when they are affected by UNPREDICTABLE results. The VAX procedure call instructions conditionally set PSL<IV> and PSL<DV>, clear PSL<FU>, and leave PSL<T> unchanged at procedure entry. (See the descriptions of the procedure-call instructions in Chapter 3 for more detail.)

The PSL is automatically saved on the stack when an exception or interrupt occurs and is saved in the process control block on a process context switch. (See Chapter 5, Interrupts and Exceptions, and Chapter 6, Process Structure, for more detail.) The PSL can also be read by the MOVPSL instruction, described in Chapter 3.

Bits <31:16> of the PSL can be changed explicitly only by executing a return from exception or interrupt (REI) instruction, described in Chapter 5. Bits <20:16> can also be changed by a move-to-processor-register (MTPR) instruction to the IPL internal processor register, as described in Chapter 5. Processor initialization sets the PSL to 041F0000 (hex).

Note that the vector processor does not use PSL<IV> and PSL<FU> to enable integer overflow and floating underflow exception checking for vector instructions. Also, vector instruction exceptions do not affect the values of the PSL condition codes bits.

1.3.3 Internal Processor Registers

The privileged internal processor register space provides access to many types of CPU control and status registers such as the memory management base registers, parts of 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. Internal processor registers are longword size, as shown in Figure 1-8b. Internal processor registers and the MTPR and MFPR instructions are described in Chapter 8.

```
31                                                              0
+---------------------------------------------------------------+
|                                                               | :Rn
+---------------------------------------------------------------+
```

a. General-Purpose Register

```
31                                                              0
+---------------------------------------------------------------+
|                                                               | :IPR n
+---------------------------------------------------------------+
```

b. Internal Processor Register

```
 3 3 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|V|M|F|I|CUR|PRV|M|         |             |D|F|I| | | | | |
|M|P|M|B|P|S|MOD|MOD|B|   IPL   |     MBZ     |V|U|V|T|N|Z|V|C|
| | | |Z|D| |   |   |Z|         |             | | | | | | | | |
+-+-+-+-+-+-+---+---+-+---------+-------------+-+-+-+-+-+-+-+-+
                                 \                           /
                                  \                         /
                                   +--------- PSW ---------+
```

c. Processor Status Longword

Figure 1-8 Processor State

Table 1-2: Fields of the Processor Status Longword

Extent	Name	Mnemonic	Meaning
<31>	Compatibility Mode	CM	When set, the processor is in PDP-11 compatibility mode (see Chapter 9). When CM is clear, the processor is in native mode. Compatibility mode may be omitted from implementations of the VAX architecture. In a processor that does not implement compatibility mode, this bit is always clear.
<30>	Trace Pending	TP	Forces a trace fault when set at the beginning of any instruction. Set by the processor if T is set at the beginning of an instruction.
<29>	Virtual-Machine Mode	VM	When set, the processor is executing a virtual machine, and the VMPSL register contains parts of the PSL of the virtual machine (see Chapter 12, Virtual Machines). When clear, the processor is running a real machine. Virtual-machine mode may be omitted from an implementation. In a processor that does not implement virtual-machine mode, VM is always clear.
<28>	Reserved		Reserved to DIGITAL; must be zero.
<27>	First Part Done	FPD	When set, execution of the instruction addressed by PC cannot simply be started at the beginning, and must be restarted at some other implementation-dependent point in its operation. If FPD is set and the exception or interrupt service routine modifies 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.
<26>	Interrupt Stack	IS	When set, the processor is executing on the interrupt stack. Any mechanism that sets IS also clears current mode and raises IPL above 0. If an REI attempts to restore a PSL with IS=1 and non-zero current mode or zero IPL, a reserved operand fault is taken. When clear, the processor is executing on the stack specified by current mode.
<25:24>	Current Access Mode	CUR_MOD	The access mode of the currently executing process. 0 Kernel 1 Executive 2 Supervisor 3 User
<22:23>	Previous Access Mode	PRV_MOD	Loaded from current mode by exceptions and CHMx instructions, cleared by interrupts, and restored by REI (see Chapter 5).
<21>	Reserved		Reserved to DIGITAL; must be zero.
<20:16>	Interrupt Priority Level	IPL	The current processor priority, in the range 0 to 1F (hex). The processor will accept interrupts only on levels greater than the current level.

1-2: Fields of the Processor Status Longword

Extent	Name	Mnemonic	Meaning
<15:8>	Reserved		Reserved to DIGITAL; must be 0.
<7>	Decimal Overflow enable	DV	When set, forces a decimal overflow trap after execution of an instruction that produced an overflowed decimal result (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.)
<6>	Floating Underflow enable	FU	When set, forces a floating underflow exception after execution of a scalar instruction that produced an underflowed result. (That is, FU forces an exception when a result exponent, after normalization and rounding, is less than the smallest representable exponent for the data type.) When FU is clear, no exception occurs.
<5>	Integer Overflow enable	IV	When set, forces an integer overflow trap after execution of a scalar 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.)
<4>	Trace enable	T	When set at the beginning of an instruction, causes TP to be set. Most programs should treat T as UNPREDICTABLE because it is set by debuggers and trace programs for tracing and for proceeding from a breakpoint. See Chapter 5 for a description of tracing.
<3>	Negative	N	When set, indicates that the last instruction that affected N produced a result that was negative. When N is clear, the result was positive or 0.
<2>	Zero	Z	When set, indicates that the last instruction that affected Z produced a result that was 0. When Z is clear, the result was non-zero.
<1>	Overflow	V	When set, indicates that the last instruction that affected V produced a result whose magnitude was too large to be properly represented in the operand that received the result or there was a conversion error. When V is clear, there was no overflow or conversion error.
<0>	Carry	C	When set, indicates that the last instruction that 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.

Change History:

Revision J. Rich Brunner, December, 1989
- o Explain what "vector" and "scalar" objects are.
- o PSL<FU>, PSL<IV>, and condition code bits not used nor affected by vector processor.
- o When the operation of the VAX scalar processor becomes UNDEFINED, so does the operation of its associated vector processor. The converse is not true.
- o Defined SBZ (Should Be Zero).

Revision H. Tim Leonard, May 1987.
- o Add virtual machines.

Revision F. Al Thomas, November 1986.
- o Add new implementation-option rules.

Revision E. Tim Leonard, September 1986.
- o Include the Introduction in Chapter One.
- o Describe wraparound from AP, FP, and SP to PC as UNPREDICTABLE.
- o Add Absolute and Self-Relative Queues.
- o Restructure PSL-fields description.

Revision 003 of the ARM. Tim Leonard, May 1985.
- o Move all the figures and tables onto separate pages.
- o Move the description of the queue data type here from Chapter 4.

Revision D. Tim Leonard, March 1985.
- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Note that octaword, D_floating, and decimal data types need not be supported in a subset implementation.
- o Move description of entire PSL to Chapter 2.
- o Move the description of instruction formats to Chapter 3.
- o Mention internal processor registers.
- o Remove a note from the description of TP. The note is wrong, and conflicts with the explanation in the Chapter 6 section about Using Trace.

Revision 7. Dileep Bhandarkar, 26 July 1982.
- o Move section on separation of procedure and data to Chapter 8.

Revision 6. Dileep Bhandarkar, 29 February 1980.
- o Add description of bit fields in registers.
- o Change trace to faults.

Revision 5. Dileep Bhandarkar, 31 January 1979.
- o Rename floating and double to F_floating and D_floating
- o Add G_floating and H_floating data types.
- o Add FF bit to PSW.

- o ECO to Overpunch alternate character set.
- o UNDEFINED must not hang.
- o MBZ checks mandatory for non-kernel mode fields, and optional for kernel mode fields.

Revision 4. Bill Strecker, 5 April 1977.

- o Replace zoned with packed.
- o Change CF to FP.
- o Add definitions of overpunch and zoned formats. Include alternate forms of overpunch.

Revision 3, ECOs 12 through 18, and results of April Task Force review. Bill Strecker, 3 June 1976.

- o Remove AL field, pointer.
- o Numeric from left separate to right zoned.
- o Remove DZ,FV.
- o IV and DV conditionally enabled by Call instructions.
- o Remove Round from C.
- o All overflowing instructions trap.
- o Remove LP.
- o Expand to 32 interrupt levels.
- o Zero length permitted for numeric and packed decimal strings.
- o Zero length field causes 0 memory references.
- o Remove 1000:1 goal and multiprocessors.
- o Change UNDEFINED result to UNPREDICTABLE.
- o Define reserved.

Revision 2, ECOs 1 through 11. Bill Strecker, 9 March 1976.

- o Clarify AL field, add pointer.
- o Add Quad.
- o Clarify floating range.
- o Don't wrap field.
- o Define nibble.
- o Combine ND,FD,ID into DZ.
- o Remove TBR.
- o C gets Round bit.
- o Registers look like memory.
- o All instructions can do I/O.

Revision 1, initial distribution. Bill Strecker, 25 September 1975.

CHAPTER 2

INSTRUCTION FORMATS AND ADDRESSING MODES

The VAX architecture has a variable-length instruction format. An instruction specifies an operation and 0 to 6 operands. An operand specifier determines how an operand is accessed. An operand specifier consists of an addressing mode specifier and, if needed, a specifier extension, immediate data, or an address, as shown in Figure 2-1a. The format of an instruction is:

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

2.1 OPCODE FORMATS

An instruction is specified by the byte address A of its opcode, as shown in Figure 2-1b. The opcode may extend over 2 bytes; the length depends on the contents of the byte at address A. Only if the value of the byte is FC (hex) through FF (hex) is the opcode 2 bytes long, as shown in Figure 2-1c.

```
                           8 7             0
            ----------------+---------------+
specifier extension, if any |addressing mode|
            ----------------+---------------+
```

a. Operand Specifier

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

b. Single-Byte Opcode

```
15             8 7             0
+---------------+---------------+
|    opcode     |    FC - FF    |   :A
+---------------+---------------+
```

c. Double-Byte Opcode

Figure 2-1 Opcodes and Operand Specifiers

2.2 OPERAND SPECIFIERS

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

The access types include:

- Read -- the specified operand is read only.
- Write -- the specified operand is written only.
- Modify -- the specified operand is read, potentially modified, and written. This is not done under a memory interlock.
- 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.
- Variable-length bit field base address -- this is the 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.
- Branch -- no operand is accessed. The operand specifier itself is a branch displacement.

The first five types are termed general mode addressing. The last type is termed branch mode addressing.

The data types include:

Byte
Word
Longword
Quadword
Octaword
F_floating
D_floating
G_floating
H_floating

The address and branch access types do not directly reference operands. For the address access type, the data type indicates the operand size to be used in the address calculation in autoincrement, autodecrement, and index modes. For the branch access type, the data type indicates the size of the branch displacement.

2.3 NOTATION

To describe the addressing modes, the following notation 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
(x)	the contents of memory location x
{ }	arithmetic parentheses for indicating 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.

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

a. Register Specifier

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

b. Register-Deferred Specifier

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

c. Autoincrement Specifier

```
               8 7      4 3      0
  --------------+--------+--------+
immediate data  |   8    |   F    |
  --------------+--------+--------+
```

d. Immediate Specifier and Extension

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

e. Autoincrement-Deferred Specifier

```
39                                                  8 7      4 3      0
+-----------------------------------------------------+--------+--------+
|               absolute address of data              |   9    |   F    |
+-----------------------------------------------------+--------+--------+
```

f. Absolute Specifier and Extension

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

g. Autodecrement Specifier

Figure 2-2 Addressing-Mode Specifiers

```
                                        15             8 7     4 3     0
                                        +---------------+-------+-------+
                                        |  byte displ   |   A   |  reg  |
                                        +---------------+-------+-------+
```

h. Byte-Displacement Specifier and Extension

```
                        23                             8 7     4 3     0
                        +-------------------------------+-------+-------+
                        |       word displacement       |   C   |  reg  |
                        +-------------------------------+-------+-------+
```

i. Word-Displacement Specifier and Extension

```
39                                                     8 7     4 3     0
+-------------------------------------------------------+-------+-------+
|                  longword displacement                |   E   |  reg  |
+-------------------------------------------------------+-------+-------+
```

j. Longword-Displacement Specifier and Extension

```
                                        15             8 7     4 3     0
                                        +---------------+-------+-------+
                                        |  byte displ   |   B   |  reg  |
                                        +---------------+-------+-------+
```

k. Byte-Displacement-Deferred Specifier and Extension

```
                        23                             8 7     4 3     0
                        +-------------------------------+-------+-------+
                        |       word displacement       |   D   |  reg  |
                        +-------------------------------+-------+-------+
```

l. Word-Displacement-Deferred Specifier and Extension

```
39                                                     8 7     4 3     0
+-------------------------------------------------------+-------+-------+
|                  longword displacement                |   F   |  reg  |
+-------------------------------------------------------+-------+-------+
```

m. Longword-Displacement-Deferred Specifier and Extension

Figure 2-2 Addressing Mode Specifiers (continued)

2.4 GENERAL MODE ADDRESSING FORMATS

Except for literal mode, an operand specifier in the general mode addressing format consists of a register number in bits <3:0> and an addressing mode specifier in bits <7:4>. The operand specifier could possibly be followed by a specifier extension, as shown in Figure 2-1a.

For a summary of general register addressing, see Table 2-2.

2.4.1 Register Mode

The register mode operand specifier format is shown in Figure 2-2a. 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, G_floating, and certain field operands). The format is as follows:

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

Because registers do not have memory addresses, the operand address is not defined and register mode may not be used for operand specifiers of address access type (except in the case of the base address for bit-field instructions, described in Chapter 3). If register mode is so used, an illegal addressing mode fault results. (See Chapter 5, Interrupts and Exceptions, for a description of faults.) 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 that takes two adjacent registers. Again, if it is used, the results are UNPREDICTABLE in the same fashion. If PC is used in register mode for a write access type operand that takes two adjacent registers, the contents of R0 are UNPREDICTABLE. If R12, R13, SP, or PC are used in register mode addressing for an operand that takes four adjacent registers, the results are UNPREDICTABLE. If PC is used in register mode for a write access type operand that requires four 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 that takes four adjacent registers, the contents of R0 are UNPREDICTABLE; and, if SP is used in register mode for a write access type operand which takes four adjacent registers, the contents of R0 and R1 are UNPREDICTABLE.

The assembler notation for register mode is Rn.

2.4.2 Register Deferred Mode

The register deferred mode operand specifier format is shown in Figure 2-2b. No specifier extension follows. In register deferred mode addressing, the address of the operand is the contents of register n:

```
        OA = Rn

        operand = (OA)
```

PC should not be used in register deferred mode addressing. If PC is used, 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).

2.4.3 Autoincrement Mode

The autoincrement mode operand specifier format is shown in Figure 2-2c. No specifier extension follows. If Rn denotes PC, immediate data follows, and the mode is termed immediate mode. See Figure 2-2d. 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 or F_floating; 8 for quadword, G_floating, or D_floating; and 16 for octaword or H_floating) is added to the contents of register n. The contents of register n is then replaced by the result:

```
        OA = Rn

        Rn <- Rn + size

        operand = (OA)
```

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

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

2.4.4 Autoincrement Deferred Mode

The autoincrement deferred mode operand specifier format is shown in Figure 2-2e. No specifier extension follows. If Rn denotes PC, a longword address follows, and the mode is termed absolute mode. See Figure 2-2f. 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)+. The notation for absolute mode is @#address, where address is the longword that follows.

2.4.5 Autodecrement Mode

The autodecrement mode operand specifier format is shown in Figure 2-2g. No specifier extension follows. In autodecrement mode addressing, the size of the operand in bytes (1 for byte; 2 for word; 4 for longword or F_floating; 8 for quadword, G_floating, or D_floating; and 16 for octaword or H_floating) is subtracted from the contents of register n. The contents of register n are then replaced by the result. The updated contents of register n is the address of the operand:

```
        Rn  <- Rn  - size

        OA = Rn

        operand = (OA)
```

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

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

2.4.6 Displacement Mode

There are three displacement mode operand specifier formats. They are termed byte displacement mode, word displacement mode, and longword displacement mode. In each, the specifier extension is a signed displacement. See Figures 2-2h, i, and j.

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. The result is the operand address:

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

        operand = (OA)
```

If Rn denotes PC, the mode is termed PC relative addressing mode. The updated contents of PC (the address of the first byte beyond the ‘,pecifier extension) is used as the base address.

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

2.4.7 Displacement Deferred Mode

The three displacement deferred mode operand specifier formats are termed byte displacement deferred mode, word displacement deferred mode, and longword displacement deferred mode. In each, the specifier extension is a signed displacement. See Figures 2-2k, l, and m.

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. The result is the address of a longword whose contents is the operand address:

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

        operand = (OA)
```

If Rn denotes PC, the mode is termed PC relative deferred addressing mode. The updated contents of PC (the address of the first byte beyond the specifier extension) is used as the base address.

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.

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

a. Literal Address Mode Specifier

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

b. Representation of a Floating-Point Number as a Literal

Figure 2-3 Literal Address Mode

Table 2-1: Floating-Point Values Representable as Literals

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

```
 31                            16 15 14            7 6   4 3     0
+--------------------------------+--+---------------+-----+-------+
|               0                | 0| 128 + exponent| fra |   0   |
+--------------------------------+--+---------------+-----+-------+
```

a. As an F_floating Number

```
 31                            16 15 14            7 6   4 3     0
+--------------------------------+--+---------------+-----+-------+
|               0                | 0| 128 + exponent| fra |   0   |
+--------------------------------+--+---------------+-----+-------+
|               0                |               0                |
+--------------------------------+--------------------------------+
 63                                                             32
```

b. As a D_floating Number

```
 31                            16 15 14                  4 3   1 0
+--------------------------------+--+---------------------+-----+-+
|               0                | 0|    1024 + exponent  | fra |0|
+--------------------------------+--+---------------------+-----+-+
|               0                |               0                |
+--------------------------------+--------------------------------+
 63                                                             33
```

c. As a G_floating Number

```
 31 29 28                      16 15 14                          0
+------+-------------------------+--+-----------------------------+
| fra  |           0             | 0|       16384 + exponent      |
+------+-------------------------+--+-----------------------------+
|               0                |               0                |
+--------------------------------+--------------------------------+
|               0                |               0                |
+--------------------------------+--------------------------------+
|               0                |               0                |
+--------------------------------+--------------------------------+
 127                                                            96
```

d. As an H_floating Number

Figure 2-4 Interpretation of a Literal

2.4.8 Literal Mode

The literal mode operand specifier format is shown in Figure 2-3a. No specifier extension follows. For operands of data type byte, word, longword, quadword, and octaword, the operand is the zero extension of the 6-bit literal field:

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

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

For operands of data type F_floating, D_floating, G_floating, and H_floating, the 6-bit literal field is composed of two 3-bit fields as shown in Figure 2-3b. The exp and fra fields are used to form an F_floating, D_floating, G_floating, or H_floating operand as shown in Figures 2-4a through 2-4d, respectively. The values that can be expressed by a floating-point literal are shown in Table 2-1.

Because there is no operand address, literal mode addressing may not be used for operand specifiers of address access type. Also, literal mode addressing may not be used for operand specifiers of write or modify access type. If literal mode is used for operand specifiers of address, modify, or write access type, an illegal addressing mode fault results. (See Chapter 5, Interrupts and Exceptions, for a description of faults.)

Literal mode addressing is a very efficient way of specifying integer values in the range 0 to 63 or the floating-point values shown in Table 2-1. Literal values outside the indicated range may be obtained by using immediate mode.

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

2.4.9 Index Mode

The index mode operand specifier format is shown in Figure 2-5. 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 indexed, literal indexed, immediate indexed, or index indexed addressing mode results in an illegal addressing mode fault (described in Chapter 5). 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 (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 normally is used 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 or F_floating; 8 for quadword, D_floating, or G_floating; and 16 for octaword or H_floating), adding BOA, and taking the result:

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

        operand = (OA)
```

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

Indexed 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 x. PC cannot be used as an index register. If PC is used, a reserved addressing mode fault occurs (described in Chapter 5). If the base operand specifier is for an addressing mode that results in register modification (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 indexed mode addressing are formed by appending the word "indexed" to the addressing mode of the base operand specifier. Following are the names and assembler notation. The index register is designated Rx to distinguish it from the register Rn in the base operand specifier.

- Register deferred indexed, (Rn)[Rx]
- Autoincrement indexed, (Rn)+[Rx]
- Autoincrement deferred indexed, @(Rn)+[Rx]

  or absolute indexed, @#address[Rx]
- Autodecrement indexed, -(Rn)[Rx]
- Byte, word, or longword displacement indexed, B^D(Rn)[Rx], W^D(Rn)[Rx], or L^D(Rn)[Rx]
- Byte, word, or longword displacement deferred indexed, @B^D(Rn)[Rx], @W^D(Rn)[Rx], or @L^D(Rn)[Rx]

2.5 BRANCH MODE ADDRESSING FORMATS

The two operand specifier formats are shown in Figure 2-6. In branch mode 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 mode addressing is A, where A is the branch address. Note that the branch address and not the displacement is used.

```
         16 15   12 11    8 7     4 3     0
  ---------+-------+-------+-------+-------+
extension, | base  | base  |   4   | index |
  if any   | mode  |  reg  |       |  reg  |
  ---------+-------+-------+-------+-------+
```

Figure 2-5 Indexed Specifier and Extension

```
 7             0
+---------------+
|     displ     |
+---------------+
```

a. Byte Displacement

```
 15                            0
+-------------------------------+
|            displ              |
+-------------------------------+
```

b. Word Displacement

Figure 2-6 Branch-Mode Specifiers

Table 2-2: Summary of General Register Addressing

Addressing Mode	Assembler Notation	Decimal	Hexadecimal	r	m	w	a	v	PC	SP	AP & FP	Indexable
General Register Addressing Mode												
literal	S^#literal	0-3	0-3	y	f	f	f	f				f
indexed	base[Rx]	4	4	y	y	y	y	y	f	y	y	f
register	Rn	5	5	y	y	y	f	y	u	uq	uo	f
register deferred	(Rn)	6	6	y	y	y	y	y	u	y	y	y
autodecrement	-(Rn)	7	7	y	y	y	y	y	u	y	y	ux
autoincrement	(Rn)+	8	8	y	y	y	y	y	p	y	y	ux
autoincrement deferred	@(Rn)+	9	9	y	y	y	y	y	p	y	y	ux
byte displacement	B^displacement(Rn)	10	A	y	y	y	y	y	p	y	y	y
byte displacement deferred	@B^displacement(Rn)	11	B	y	y	y	y	y	p	y	y	y
word displacement	W^displacement(Rn)	12	C	y	y	y	y	y	p	y	y	y
word displacement deferred	@W^displacement(Rn)	13	D	y	y	y	y	y	p	y	y	y
longword displacement	L^displacement(Rn)	14	E	y	y	y	y	y	p	y	y	y
longword displacement deferred	@L^displacement(Rn)	15	F	y	y	y	y	y	p	y	y	y
Program Counter Addressing Mode												
immediate	I^#constant	8	8	y	u	u	ud	y				u
absolute	@#address	9	9	y	y	y	y	y				y
byte relative	B^address	10	A	y	y	y	y	y				y
byte relative deferred	@B^address	11	B	y	y	y	y	y				y
word relative	W^address	12	C	y	y	y	y	y				y
word relative deferred	@W^address	13	D	y	y	y	y	y				y
longword relative	L^address	14	E	y	y	y	y	y				y
longword relative deferred	@L^address	15	F	y	y	y	y	y				y

Key:
base - any indexable addressing mode
f - reserved addressing mode fault
p - Program Counter addressing
u - UNPREDICTABLE
ud - UNPREDICTABLE for destination of CALLS, CALLG, JMP, and JSB
uq - UNPREDICTABLE for quadword, octaword, D_floating, G_floating, and H_floating (and field if position plus size greater than 32)
uo - UNPREDICTABLE for octaword and H_floating
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

IMPLEMENTATION NOTE

\The preferred implementation for the UNPREDICTABLE addressing modes is to treat them as reserved addressing modes. The software must always expect them to remain UNPREDICTABLE.\

2.6 INSTRUCTION INTERPRETATION

The VAX architecture assumes a sequential control-flow model of instruction stream processing. In this model, instructions are processed from the instruction stream, which resides in instruction memory, one at a time and no instruction is processed (even partially) until the one before it completes.

The processor in interpreting an instruction performs the following three steps:

1. Reads and evaluates each operand specifier in order of instruction stream occurrence as follows:

   a. If access type is read: evaluates the operand address; if the operand is in a register, reads the operand and saves it; if the operand is not in a register, does the reading and saving of the operand anytime before performing step 2.

   b. If access type is write: evaluates the operand address and saves it.

   c. If access type is modify: evaluates the operand address and saves it; if the operand is in a register, reads the operand and saves it; if the operand is not in a register, does the reading and saving of the operand anytime before performing step 2.

   d. If access type is address: evaluates the address and saves it.

   e. If access type is branch: saves the operand specifier.

   The read accesses for operands in registers are done as the operand specifiers are processed. The bytes for non-register operands may be read in any order anytime before performing step 2. (The order in which one processor reads bytes for non-register operands within a single instruction may differ from the order in which a second processor wrote them. See section 7.1.2, the Ordering of Reads, Writes, and Interrupts.)

2. Performs the operation indicated by the instruction.

3. Stores the result(s) using the saved addresses.

   Within an operand, the result bytes may be stored in any order. Also, the result operands may be stored in any order with one restriction: if multiple results overlap, the order of operand storage is indicated by the occurrence of the corresponding operand specifiers in the instruction stream. For example, if an instruction references two operands of write or modify access type at the same address, the first will be overwritten by the second.

The implications of this instruction interpretation process are:

- 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.

- Other than as indicated in the note below, all input operands are read, and all addresses of output operands computed before any results of the instruction are stored.

- 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 multiprocessor synchronization. (See section 7.1.3, Atomicity and Corruption of Shared Data.)

NOTE

The string instructions are an exception to this sequence performed by each instruction. Partial results are stored before the instruction operation is completed.

The variable-length bit field instructions treat the position, size, and base address operand specifiers as the specification of an implied field operand specifier.

If multiple exceptions occur, the order in which they are taken is UNPREDICTABLE (except for the trace fault--see section 5.4.5). 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.

Change History:

Revision J. Rich Brunner, Tom Eggers, December 1989.
- o Input memory operands may be read by an instruction in any order.

Revision H. Tim Leonard, May 1987.

Revision E. Tim Leonard, September 1986.
- o Reformat the text, figures, and tables.

Revision D1. Tim Leonard, January 1986.
- o Prohibit CALLG, CALLS, JMP, JSB call to immediate-mode destination.

Revision D. Tim Leonard, March 1985
- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Start this section with the description of Instruction Formats from Chapter 2.
- o Make immediate indexed mode UNPREDICTABLE.

Revision 8, typos and clarifications. Dileep Bhandarkar, 26 July 1982.
- o No substantive changes.

Revision 7, unpredictable preferred action. Tom Eggers, 5 May 1980.
- o Preferred implementation for unpredictable address modes is a reserved addressing mode fault.

Revision 6, add H_floating data type. Dileep Bhandarkar, 14 July 1978.
- o Add G_floating and H_floating data types.
- o Add octaword data type.

Revision 5, editorial changes. Bill Strecker, 7 February 1977.
- o No substantive changes.

Revision 4, address mode changes. Bill Strecker, 3 June 1976.
- o Change floating literals.
- o Remove argument mode, local mode and post-index mode.
- o Add byte, word, longword displacement deferred mode.
- o Add autoincrement deferred mode.
- o Writing immediate mode UNPREDICTABLE.
- o Register deferred of PC UNPREDICTABLE.
- o PC index register gives reserved operand fault (future escape).
- o SP index register OK.
- o Same register for base and index OK for modes other than autoincrement, autodecrement, and autoincrement deferred.

- o Write to PC of operand taking 2 registers result in R0 UNPREDICTABLE.

Revision 3, ECOs 12 through 18, and results of April Task Force review. Bill Strecker, 12 May 1976.

- o Note that modify is not under memory interlock.
- o Change pointer to longword, remove <27:0> and EAL.
- o Change R'R+1 to R[n+1]'Rn (typo).
- o Clarify what is UNPREDICTABLE.
- o Modify and write of immediate is UNPREDICTABLE.
- o Change hex modes to decimal; add table.
- o Change R to PL in local mode (typo).
- o Clarify value of PC in displacement and branch displacement.
- o Change address mode abort to address mode fault.

Revision 2, ECOs 1 through 11. Bill Strecker, 10 March 1976.

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

Revision 1, initial distribution. Bill Strecker, 25 September 1975.

CHAPTER 3

INSTRUCTIONS

This chapter describes the instructions generally used by all software, across all implementations of the VAX architecture. Certain instructions are specific to portions of the VAX architecture: memory management, interrupts and exceptions, process dispatching, and processor registers. Such instructions are generally used by privileged software and are described in chapters devoted to those portions of the architecture. A concise list of opcode assignments appears in Appendix A.

The instruction set is divided into 12 major sections:

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

Within each major section, closely related instructions are grouped 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. The format presents the name and type of each instruction operand specifier and the order in which it appears in memory. Operand specifiers from left to right appear in increasing memory addresses. Implied operands are enclosed by { }.

3. The operation of the instruction. The operation is given as a sequence of pseudo-code statements in an ALGOL-like syntax. Each VAX processor may implement the instruction in different

or more efficient ways, but each processor gives results consistent with the pseudo-code, English descriptions, and notes.

4. The effect on condition codes.

5. Exceptions specific to the instruction. Exceptions generally possible for all instructions are not listed (for example, illegal or reserved addressing mode, trace, and memory management exceptions).

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. In many cases, notes on the instruction and programming examples.

3.1 NOTATION

Operand specifiers are described in the following way:

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

The name is suggestive of the operand in the context of the instruction. The name is often abbreviated. The access type is represented by a letter denoting the operand specifier access type. These are:

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

b No operand reference. The operand specifier is a branch displacement. The size of branch displacement is given by <data type>.

m Operand is read, potentially modified, and written. Note that this is not an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessibility (see Chapter 4, Memory Management).

r Operand is read only.

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

w Operand is written only.

The data type in the operand specifier notation 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
*	multiple longwords

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 2.

+	addition
-	subtraction, unary minus
*	multiplication
/	division (quotient only)
**	exponentiation
'	concatenation
<-	is replaced by
=	is defined as

Rn or R[n]	contents of register Rn
PC, SP, FP, or AP	the contents of register R15, R14, R13, or R12 respectively
PSW	the contents of the processor status word
PSL	the contents of the processor status longword
(x)	contents of memory location whose address is x
(x)+	contents of memory location whose address is x; x incremented by the size of operand referenced at x
-(x)	x decremented by size of operand to be referenced at x; contents of memory location whose address is new value of x
<x:y>	a modifier that delimits an extent from bit position x to bit position y inclusive
<x1,x2,...,xn>	a modifier that enumerates bits x1,x2,...,xn
{ }	arithmetic parentheses used to indicate precedence
AND	logical AND
OR	logical OR
XOR	logical XOR
NOT	logical (one's) complement
LSS	less than signed
LSSU	less than unsigned
LEQ	less than or equal signed
LEQU	less than or equal unsigned
EQL	equal signed
EQLU	equal unsigned
NEQ	not equal signed
NEQU	not equal unsigned
GEQ	greater than or equal signed
GEQU	greater than or equal unsigned
GTR	greater than signed

GTRU	greater than unsigned
SEXT(x)	x is sign extended to size of operand needed
ZEXT(x)	x is zero extended to size of operand needed
REM(x,y)	remainder of x divided by y, such that x/y and REM(x,y) have the same sign
MINU(x,y)	minimum unsigned of x and y
MAXU(x,y)	maximum unsigned of x and y

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; whereas "+" applied to byte operands is an integer byte add. Similarly, "LSS" is a floating comparison when applied to floating operands; whereas "LSS" is an integer byte comparison when applied to byte operands.

4. Instruction operands are evaluated according to the operand specifier conventions (see the description of instruction interpretation starting on page 2-18). 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, two 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.

3.2 INTEGER-ARITHMETIC AND LOGICAL INSTRUCTIONS

ADAWI Add Aligned Word Interlocked

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

reserved operand fault
integer overflow

Opcode:

```
  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. If the sum operand is contained in memory, then the operation is interlocked against interlocked operations to the same address from other processors. The destination must be aligned on a word boundary (bit 0 of the address of the sum operand must be zero). If it is not, a reserved operand fault is taken.

Notes:

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

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

ADD Add

Format:

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

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

Operation:

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

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

Condition Codes:

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

Exception:

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 PSL<C> and the addend operand are added to the sum operand, and the sum operand is replaced by the result.

Notes:

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

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

ASH Arithmetic Shift

Format:

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

Operation:

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

Condition Codes:

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

Exception:

integer overflow

Opcodes:

78	ASHL	Arithmetic Shift Long
79	ASHQ	Arithmetic Shift Quad

Description:

The source operand is arithmetically shifted by the number of bits specified by the count operand, and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand shifts to the left bringing zeros into the least significant bit. A negative count operand shifts to the right bringing 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 0;
        Z <- dst EQL 0;
        V <- 0;
        C <- C;
```

Exceptions:

none

Opcodes:

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

Description:

In 2 operand format, the destination operand is ANDed with the one's 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 one's complement of the mask operand and the destination operand is replaced by the result.

BIS   Bit Set

Format:

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

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

Operation:

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

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

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

Description:

The destination operand is replaced by 0.

Notes:

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

2. The CLRO instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

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;
```

Exception:

integer overflow

Opcodes:

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

Description:

The source operand is converted to the data type of the destination operand, and the destination operand is replaced by the result. Conversion of a shorter data type to a longer one is done by sign extension; conversion from longer to 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};
```

Exception:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

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

DIV Divide

Format:

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

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

Operation:

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

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

Condition Codes:

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

Exceptions:

integer overflow
divide by zero

Opcodes:

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

Description:

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

Notes:

1. The remainder, if any, is lost.

2. Division is performed such that the remainder (unless it is 0) has the same sign as the dividend; that is, the result is truncated toward 0.

3. Integer overflow occurs if and only if the largest negative integer is divided by -1. On overflow, operands are affected as in item 4 below.

4. 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. \If execution of DIVL2 results in overflow, the quotient is UNPREDICTABLE.\

EDIV Extended Divide

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

integer overflow
divide by zero

Opcodes:

7B	EDIV	Extended Divide

Description:

The dividend operand is divided by the divisor operand; the quotient operand is replaced by the quotient; and the remainder operand is replaced 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 item 3 below.

3. If the divisor operand is 0, then the quotient operand is replaced by bits <31:0> of the dividend operand; the remainder operand is replaced by 0.

EMUL   Extended Multiply

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcode:

```
  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};
```

Exception:

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 one's 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;
```

Exception:

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	MOVO	Move Octa

Description:

The destination operand is replaced by the source operand.

Notes:

1. The MOVO instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

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 0. 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;     !2 operand

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

Condition Codes:

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

Exception:

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  {-(SP).wl}
```

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcode:

DD	PUSHL	Push Long

Description:

The longword source operand is pushed on the stack.

Notes:

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

ROTL Rotate Long

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcode:

9C	ROTL	Rotate Long

Description:

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

SBWC Subtract With Carry

Format:

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

Operation:

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

Condition Codes:

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

Exception:

integer overflow

Opcode:

```
  D9    SBWC    Subtract With Carry
```

Description:

The subtrahend operand and the contents of PSL<C> are subtracted from the difference operand, and the difference operand is replaced by the result.

Notes:

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

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

SUB Subtract

Format:

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

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

Operation:

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

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

Condition Codes:

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

Exceptions:

integer overflow

Opcodes:

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

Description:

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

Notes:

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

TST Test

Format:

```
        opcode src.rx
```

Operation:

```
        src - 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:

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

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

Operation:

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

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

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

Change History:

Revision J. Rich Brunner, December 1989.

- o Add pseudo-code clarification.
- o ADAWI interlocks only when sum operand in memory.

Revision H. Tim Leonard, May 1987.

Revision F. Al Thomas, November 1986.
Add new instruction-implementation options.

Revision E. Al Thomas, September 1986.

- o Add implied operand for PUSHL.
- o Include CLRO and MOVO as optional instructions.
- o Overflow during DIVL2 will result in an UNPREDICTABLE quotient.

Revision D. Tim Leonard, March 1985.

- o Change the revision number to correspond to DEC Standard 032 rev number.

Revision 7. Dileep Bhandarkar, 26 July 1982.
Revision 6. Dileep Bhandarkar, 27 March 1980.

- o Add variable bit field address access type. Change .ab to .vb.
- o Clarify condition codes for BPT, HALT.

Revision 5, octaword ECO. Dileep Bhandarkar, 27 October 1978.

- o Add G_floating, H_floating, and Octaword data types.
- o Add MOVO and CLRO(same as CLRH).
- o Fix variable bit field description.
- o Add interlocked self-relative queue instructions: INSQHI, INSQTI, REMQHI, REMQTI.
- o Change INSV condition codes.

Revision 4. Bill Strecker, 12 March 1977.

- o Add ADAWI.
- o Correct opcode assignments.
- o CF to FP.
- o CALL Standard ECO.
- o Add INDEX instruction.
- o Expand descriptions of XFC, HALT, and BPT.
- o Add motivation for call frame.
- o Clear T on CALL.
- o Reverse incorrect opcode assignments of FFC, FFS.

Revision 3, ECOs 12 through 18, results of April Task Force review and May 25 meeting. Bill Strecker, 10 June 1976.

- o Change FFC and FFS condition codes.
- o Change BLS and BLC to BLBS and BLBC.
- o Change number of bytes referenced on access to zero length field from 1 to zero.
- o Change HALT in non-kernel mode to privileged instruction fault.

- Change BPT to breakpoint fault.
- Change MOVPSW to MOVPSL.
- Add queue instructions.
- Add MINU function in ISP.
- Explicitly give SEXT or ZEXT in all cases needed.
- Specified condition codes on all exceptions.
- Remove CMPA, DIFA, ADTA, SBFA.
- Include pos<31> in Field.
- Correct operand type typos in field.
- Add BBSSI, BBCCI.
- In ACBx, SOBxx, AOBxx set N, Z, V from the add.
- Change names to AOBLEQ, AOBLSS, SOBGEQ, SOBGTR.
- Specify branch behavior on overflow ACBx, AOBxxx, SOBxxx.
- In CALLS and CALLG remove DZ and FV, set DV and IV from entry mask, clear FU, remove ring crossing.
- BISPSW and BICPSW take reserved operand fault.
- Remove MSx, MSPx.
- Reserved operand aborts become faults.
- C <- 0 for ASHL, ASHQ.
- Change pointer to longword or address; make it 32 bits.
- Add MOVQ, CLRQ, MOVZBW.
- Change CMP condition codes, and conditional branches, per ECO 17.
- Change results on overflow in EDIV.
- Specify condition codes on all exceptions.
- Split into separate specifications.

Revision 2, ECOs 1 through 11. Bill Strecker, 16 March 1976.

- Qualified operand names used in instruction description.
- BLISS relational operators used in instruction operation.
- Detailed operation descriptions included on all instructions.
- Condition code settings specified for all instructions.
- Eliminate PUSH {B,W,F,D}, POP {B,W,L,F,D}, EXCH {B,W,L,F,D}, MOVM {B,W,L,F,D}, INS {B,W,L}, ADWC {B,W}, SBWC {B,W}, MOD {B,W,L}, BFC, AND {B,W,L}, ASH {B,W}, USH {B,W,L}, ROT {B,W}, ROTC {B,W,L}, SXT {B,W,L}, ADDC {F,D}, SUBC{F,D}, MULC{F,D}, DIVC{F,D}, INC {F,D}, DEC {F,D}, AVP, BSP, CASE{B,W} {B,W,L}, CALL, SCB, THRD, LSTZ, LSTO, ASP, MOVRJS, CMPRLS, SCNRLS, LOCRLS, MODN.
- Add ASHQ, CVTR{F,D} L, CHOP{F,D}, POLY{F,D}, ADTA, SBFA, DIFA, EXTV, CMPV, FFS, FFC, BPN, BME, BRW, BSBW, BLS, BLC, CASE{B,W,L}, CALLS, CALLG, AOBLT, SOBGT, JMP, JSB, MOVPSW, HALT, MS{B,W,L,F,D,Q}, MPS{B,W,L=F,D=Q}, MOVTUC, SPANC, MATCHC, 6 operand ADDN, 6 operand SUBN, 6 operand MULN, 6 operand DIVN.
- Change MOVN {B,W,L,F,D} to MNEG {B,W,L,F,D}, MOVC {B,W,L} to MCOM {B,W,L}, MOVZ {B,W} to MOVZ {BW,BL,WL}, MOVA{B,W,L=F,D} to MOVP{B,W,L=F,D=Q}, PUSHA{B,W,L=F,D} to PUSHP{B,W,L=F,D=Q}, BR to BRB, BSB to BSBB, AOB to AOBLE, SOB to SOBGE, MOVLJS to MOVC3 and MOVC5, MOVTS to MOVTC, CMPLRS to CMPC3 and CMPC5, SCNLRS to SCANC, LOCLRS to LOCC, EDITS to EDITPC, CVTLN to CVTLS and CVTLU, CVTPN to CVTPS and CVTPU, CVTSN and CVTUN to ASHS and ASHU, MOVN to MOVS and MOVU.
- Change MULX to EMUL, DIVX to EDIV, MI {F,D} to EMOD {F,D}, CMPA{B,W,L=F,D} to CMPA, MOVZV to EXTZV, CMPV to CMPZV,

RETURN to RET. The instruction operands or operations changed.
- ACB redefined.
- C <- 0 for MUL {B,W,L}.
- ISP added for string instructions.
- Numeric suffix added to indicate number of operands (ADDB3, for example).

Revision 1, initial distribution. Bill Strecker, 25 September 1975.

3.3 ADDRESS INSTRUCTIONS

MOVA Move Address

Format:

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

Operation:

```
        dst <- src;
```

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

Notes:

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

2. The MOVAH and MOVAO instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

PUSHA Push Address

Format:

```
opcode src.ax  {-(SP).wl}
```

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.

3. The PUSHAH and PUSHAO instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

Change History:

Revision J. Rich Brunner, December 1989.

Revision H. Tim Leonard, May 1987.

Revision F. Al Thomas, November 1986.
- o Add new instruction-implementation rules.

Revision E. Al Thomas, September 1986.
- o Add implied operand for PUSHA.
- o Include MOVAH, MOVAO, PUSHAH, PUSHAO as optional instructions.

Revision D. Tim Leonard, March 1985.
- o Change the revision number to correspond to DEC Standard 032 rev number.

Revision 7, repartition Chapter 4. Dileep Bhandarkar, 26 July 1982.

3.4 BIT-FIELD INSTRUCTIONS

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

1. A longword position operand.

2. A byte field-size operand that 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. Unlike other instances of operand specifiers of address access type, however, 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 1). 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 bit-field instructions, a macro FIELD(pos, size, address) is introduced with the following expansion (if size NEQ 0):

```
FIELD(pos, size, address)

= (address + SEXT(pos<31:3>))<{size - 1} + pos<2:0>:pos<2:0>>
      !if address not specified by register mode

= {R[n+1]'Rn}<{size - 1} + pos:pos>
      !if address specified by register mode and pos + size
      !GTRU 32

= Rn<{size - 1} + pos:pos>
      !if address specified by register mode and pos + size
      !LEQU 32
```

In general, the number of bytes referenced by the contents ( ) operator above is:

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

However, in the case of INSV, it may reference more than this, up to and including the entirety of the aligned longword(s) that contains the field.

Zero bytes are referenced if the field size is 0.

CMP Compare Field

Format:

```
        opcode pos.rl, size.rb, base.vb, {field.rv}, src.rl
```

Operation:

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

Exception:

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:

   o size GTRU 32, or

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

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

EXT Extract Field

Format:

```
        opcode pos.rl, size.rb, base.vb, {field.rv}, dst.wl
```

Operation:

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

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

Condition Codes:

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

Exception:

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 actions are to replace the destination operand with 0 and to affect the condition codes.

Notes:

1. A reserved operand fault occurs if:

   o size GTRU 32, or

   o pos GTRU 31, and 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, {field.rv}, 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;
```

Exception:

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 PSL<Z> 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 PSL<Z> is set. If the size operand is 0, the find position operand is replaced by the start position operand and PSL<Z> is set.

Notes:

1. A reserved operand fault occurs if:

   - size GTRU 32, or

   - startpos GTRU 31, and 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, {field.mv}
```

Operation:

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

Condition Codes:

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

Exception:

reserved operand

Opcode:

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 instruction has no effect.

Notes:

1. When executing INSV, a processor may read in the entire aligned longword(s) that contains the field, replace the field portion of the aligned longword(s) with the source operand, and write back the entire aligned longword(s). Because of this, data written to the non-field portion of the aligned longword(s) in memory by another processor or I/O device during the execution of the INSV may be written over when the INSV is completed.

2. A reserved operand fault occurs if:

   o size GTRU 32, or

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

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

Change History:

Revision J. Rich Brunner, December 1989.

- o Add to INSV description that INSV can write the entirety of the aligned longword(s) which contains the field. Added the same to description of the contents operator () of the FIELD macro.

Revision H. Tim Leonard, May 1987.

Revision E. Al Thomas, September 1986.

- o Add implied operands for instructions.

Revision D. Tim Leonard, March 1985.

- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Change the description of INSV from .wv to .mv.

Revision 7, repartition Chapter 4. Dileep Bhandarkar, 26 July 1982.

- o Create separate file for this section.
- o INSV with size = 0 is a no op.

3.5 CONTROL INSTRUCTIONS

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

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 overflow};
        C <- C;
```

Exceptions:

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

Opcodes:

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

Description:

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

Notes:

1. ACB directly 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 PSL<FU> is clear, the index operand is replaced by 0 and comparison and branch determination proceed normally. A fault occurs if PSL<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. The ACBx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

7. The ACBx instructions are in the emulated-only group and are unlikely to be implemented in future VAX processors. Software developers are advised to avoid using these instructions.

AOBLEQ Add One and Branch Less Than or Equal

Format:

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

Operation:

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

Condition Codes:

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

Exception:

integer overflow

Opcode:

F3	AOBLEQ	Add One and Branch Less Than or Equal

Description:

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

Notes:

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

2. The C-bit is unaffected.

AOBLSS Add One and Branch Less Than

Format:

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

Operation:

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

Condition Codes:

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

Exception:

integer overflow

Opcode:

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; thus the branch is taken (unless the limit operand is the largest negative integer).

2. The C-bit is unaffected.

B Branch on (condition)

Format:

```
        opcode displ.bb
```

Operation:

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

Condition Codes:

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

Exceptions:

none

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

Description:

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

Notes:

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

1. Overflow and Carry Group

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

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

2. Unsigned Group

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

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

3. Signed Group

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

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

BB Branch on Bit

Format:

```
        opcode pos.rl, base.vb, displ.bb, {field.rv}
```

Operation:

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

Condition Codes:

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

Exception:

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 the section "Bit-Field Instructions" earlier in this chapter for a definition of FIELD.

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

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

BB Branch on Bit (and modify without interlock)

Format:

```
        opcode pos.rl, base.vb, displ.bb, {field.mv}
```

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;
```

Exception:

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 the section "Bit-Field Instructions" earlier in this chapter for a definition of FIELD.

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

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

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

BB Branch on Bit Interlocked

Format:

```
        opcode pos.rl, base.vb, displ.bb, {field.mv}
```

Operation:

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

Condition Codes:

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

Exception:

reserved operand

Opcodes:

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

Description:

The single-bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to 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 the section "Bit-Field Instructions" earlier in this chapter for a 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$:     BBSSI   bit, base, 2$    ; Try to obtain interlock.
                                 ; Fall through if successful,
                                 ; branch to 2$ if unsuccessful.

; Insert the code sequence that requires the interlock here.

        BRB     3$               ; Branch around busy loop.

; Wait for the busy interlock to become free, using an
; non-interlocked instruction.  When the lock becomes free, try the
; interlocked instruction again.

2$:     BBS     bit, base, 2$    ; Loop until lock becomes free.
        BRB     1$               ; Try interlocked reference again.

3$:     BBCCI   bit, base, 4$    ; Release the interlock.
4$:
```

\This example replaced a previous example, "1$: BBSSI bit,base,1$", for performance reasons. Looping on the BBSSI instruction creates an interlocked memory read-write sequence each time the instruction executes, which may degrade performance for other processors. The new example loops on an non-interlocked instruction which has lower overhead, particularly if the data reference is satisfied by a local cache. When the interlock bit is finally cleared, the cache location is invalidated, the BBS falls out of the loop, and the interlocked instruction is tried again.\

BLB Branch on Low Bit

Format:

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

Operation:

```
teststate = if {BLBS} then 1 else 0;
if src<0> EQL teststate then
        PC <- PC + SEXT(displ);
```

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

The low bit (bit 0) of the source operand is tested and, if it is equal to the test state indicated by the instruction, the sign-extended branch displacement is added to PC. PC is then 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, {-(SP).wl}
```

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

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

CASE   Case

Format:

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

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

The base operand is subtracted from the selector operand, and a temporary is replaced by the result. The temporary is compared with the limit operand; if it is less than or equal unsigned, a branch displacement selected by the temporary value is added to PC. PC is then replaced by the result. Otherwise, two 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 at 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.

3. The Pascal statement:

```
case i of
     32:                x := sin(x);
     33:                x := cos(x);
     34:                x := exp(x);
     35:                x := ln(x);
     36, 37:            x := arctanh(x);
     otherwise          x := reserved
end
```

is translated by the VAX Pascal compiler to:

```
            casel   i, #32, #<37-32>
    1$:     .word   sin - 1$          ; Selector is 32.
            .word   cos - 1$          ; Selector is 33.
            .word   exp - 1$          ; Selector is 34.
            .word   ln - 1$           ; Selector is 35.
            .word   arctanh - 1$      ; Selector is 36.
            .word   arctanh - 1$      ; Selector is 37.
    otherwise:
            movl    reserved, x       ; Selector is less than
                                      ; 32 or greater than 37.
```

JMP Jump

Format:

```
        opcode dst.ab
```

Operation:

```
        PC <- dst;
```

Condition Codes:

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

Exceptions:

        none

Opcode:

```
  17    JMP     Jump
```

Description:

PC is replaced by the destination operand.

Notes:

If the destination operand's addressing mode is immediate mode, the results of the instruction are UNPREDICTABLE.

JSB Jump to Subroutine

Format:

```
opcode dst.ab, {-(SP).wl}
```

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:

1. 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)+.

2. If the destination operand's addressing mode is immediate mode, the results of the instruction are UNPREDICTABLE.

RSB Return from Subroutine

Format:

```
        opcode {(SP)+.rl}
```

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.

SOBGEQ Subtract One and Branch Greater Than or Equal

Format:

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

Operation:

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

Condition Codes:

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

Exception:

integer overflow

Opcode:

F4	SOBGEQ	Subtract One and Branch Greater Than or Equal

Description:

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

Notes:

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

2. The C-bit is unaffected.

SOBGTR Subtract One and Branch Greater Than

Format:

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

Operation:

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

Condition Codes:

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

Exception:

integer overflow

Opcode:

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

Description:

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

3. \VAX/VMS uses

```
        1$:     SOBGTR  COUNT, 1$
```

   as a calibrated timing delay. \

Change History:

Revision J. Rich Brunner, December 1989.

Revision H. Tim Leonard, May 1986.
- o Change the example busy wait loop in BBSSI.

Revision F. Al Thomas, November 1986.
- o Add new instruction-implementation rules.

Revision E. Al Thomas, September 1986.
- o Add implied operands for respective instructions.
- o Include ACBF, ACBD, ACBG, and ACBH as optional instructions.

Revision D1. Tim Leonard, January 1986.
- o Prohibit JMP and JSB to immediate-mode destination.

Revision D. Tim Leonard, March 1985.
- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Add an example of use to the description of CASE.

Revision 7, repartition Chapter 4. Dileep Bhandarkar, 26 July 1982.
- o MicroVAX implements ACBF and ACBG.

3.6 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. The CALLG instruction calls a procedure with the argument list in an arbitrary location. The CALLS instruction calls a procedure with the argument list on the stack. Upon return after a CALLS, 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, as shown in Figure 3-1. 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 intermodule calls to major VAX software subsystems comply with the procedure-calling software standard. 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, shown in Figure 3-2. This structure contains the saved registers, the saved PSW, the register save mask, and several control bits. The frame also includes a longword that the CALL instructions clear; this is used to implement the VAX/VMS condition-handling facility. Refer to the VAX/VMS document set. 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. 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.

*Refer to the VAX/VMS document set for the procedure-calling standard.

```
 15 14 13 12 11                        0
+--+--+-----+-----------------------+
|DV|IV| MBZ |       registers       |
+--+--+-----+-----------------------+
```

Figure 3-1 Procedure Entry Mask

```
 31 30 29 28 27                16 15 14                5 4       0
+-------------------------------------------------------------------+
|              condition handler (initially 0)                      | :(FP)
+-----+--+--+--------------------+--+--------------------+-----------+
| SPA | S| 0|      mask<11:0>    | Z| saved PSW<14:5>   |     0     |
+-----+--+--+--------------------+--+--------------------+-----------+
|                          saved AP                                 |
+-------------------------------------------------------------------+
|                          saved FP                                 |
+-------------------------------------------------------------------+
|                          saved PC                                 |
+-------------------------------------------------------------------+
|                          saved R0 (...)                           |
+-------------------------------------------------------------------+
       .                                                   .
       .                                                   .
       .                                                   .
+-------------------------------------------------------------------+
|                          saved R11 (...)                          |
+-------------------------------------------------------------------+
    (0 to 3 bytes specified by SPA, Stack Pointer Alignment)

        S = Set if CALLS; clear if CALLG.
        Z = Always cleared by CALL. Can be set by software to
            force a reserved operand fault on a RET.
```

Figure 3-2 Procedure-Call Stack Frame

CALLG Call Procedure with General Argument List

Format:

```
        opcode arglist.ab, dst.ab, {-(SP).w*}
```

Operation:

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

Condition Codes:

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

Exception:

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 bits <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 following is pushed on the stack: 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>, a 0 in bit <15> and PSW<14:0> in bits <14:0> with T and the condition codes cleared. 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>, respectively, of the entry mask; floating underflow is cleared. The 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.

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 that are modified in the called procedure must be preserved in the mask.

4. The alignment bytes left on the stack are UNPREDICTABLE. They may, for example, be written with zeros when the stack is aligned.

5. If the destination operand's addressing mode is immediate mode, the results of the instruction are UNPREDICTABLE.

CALLS Call Procedure with Stack Argument List

Format:

```
        opcode numarg.rl, dst.ab, {-(SP).w*}
```

Operation:

```
        tmp2 <- (dst);
        -(SP) <- numarg;
        tmp1 <- SP;
        SP<1:0> <- 0;

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

Condition Codes:

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

Exception:

reserved operand

Opcode:

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 following is pushed on the stack: 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>, a 0 in bit <15> and PSW<14:0> in bits <14:0> with T and the condition codes cleared. 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>, respectively, of the entry mask; 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.

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 that are modified in the called procedure must be preserved in the entry mask.

5. The alignment bytes left on the stack are UNPREDICTABLE. They may, for example, be written with zeros when the stack is aligned.

6. If the destination operand's addressing mode is immediate mode, the results of the instruction are UNPREDICTABLE.

7. In order to facilitate high speed implementations of the CALLS instruction, a restriction is placed on addressing mode combinations involving logically inconsistent simultaneous use of a value as both a numeric value and an address. Specifically, if the content of register Rn is used as the numarg operand, and the dst specifier uses the contents of Rn as an address in an addressing mode that modifies Rn (autodecrement, autoincrement, or autoincrement deferred modes), the result is UNPREDICTABLE. For the same reason, if the content of SP (which is implicitly modified) is used as the numarg operand, the result is UNPREDICTABLE.

RET Return from Procedure

Format:

```
        opcode {(SP)+.r*}
```

Operation:

```
        SP <- FP + 4;
        tmp1 <- (SP)+;
        AP <- (SP)+;
        FP <- (SP)+;
        PC <- (SP)+;
        tmp2 <- tmp1<27:16>;
        for tmp3 <- 0 step 1 until 11 do
                if tmp2<tmp3> EQL 1 then
                R[tmp3] <- (SP)+;

        SP <- SP + ZEXT{tmp1<31:30>}

        if tmp1<29> EQL 1 then
                begin
                tmp4 <- 4 * ZEXT({(SP)+}<7:0>);
                SP <- SP + tmp4;
                end;

        PSW <- tmp1<15:0>;
```

Condition Codes:

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

Exception:

reserved operand

Opcode:

```
  04    RET     Return from Procedure
```

Description:

SP is replaced by FP plus 4. A longword containing the following is popped from the stack and saved in a temporary: stack alignment bits in bits <31:30>, a flag distinguishing CALLS from CALLG in bit <29>, the low 12 bits of the procedure entry mask in bits <27:16>, and a saved PSW in bits <15:0>. 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 tmp1<15:8> NEQ 0.

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

3. The value of tmp1<28> is ignored.

4. The procedure-calling standard and condition-handling facility assume that procedures returning a function value or a status code do so in R0 or R0 and R1.

5. If FP<1:0> is not zero, or if the stack frame is ill-formed, the results are UNPREDICTABLE. \This allows an implementation to OR the alignment bits with SP rather than to add them.\

6. \The longword addressed by FP at the beginning of the instruction is the condition handler address. It is not used by the RET instruction, and whether RET references it or not is UNPREDICTABLE.\

NOTE TO IMPLEMENTORS

\If RET is faulted or interrupted, SP must be restored to the original value. Since RET writes SP but does not read it, this would seem to be an unnecessary restriction. Even if RET modified SP and failed to restore it, the instruction would complete correctly after being restarted. However, the fault or interrupt may cause an AST to be delivered. If SP now points into the stack, the stack will be destroyed by delivery of the AST. To keep this from happening, SP must be restored to the original value when RET faults.\

Change History:

Revision J. Rich Brunner, December 1989.

Revision H. Tim Leonard, April 1987.

Revision F. Al Thomas, November 1986.

- o Add restrictions for addressing modes of CALLS operands.

Revision E. Al Thomas, September 1986.

- o Add implied operands to instruction descriptions.

Revision D1. Tim Leonard, January 1986.

- o Prohibit CALLx to an immediate-mode destination.

Revision D. Tim Leonard, March 1985.

- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Refer to VAX/VMS Introduction to System Routines for calling standard.
- o RET may or may not reference the longword FP points to.
- o Alignment bytes are UNPREDICTABLE.
- o Reserve stack frame bit (FP+4)<15> as MBZ forever.

Revision 7, repartition Chapter 4. Dileep Bhandarkar, 26 July 1982.

3.7 MISCELLANEOUS INSTRUCTIONS

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>};
```

Exception:

reserved operand

Opcode:

B9	BICPSW Bit Clear PSW

Description:

PSW is ANDed with the one's 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>;
```

Exception:

reserved operand

Opcode:

```
  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

Format:

```
        opcode  {-(KSP).w*}
```

Operation:

```
        {initiate breakpoint fault};
```

Condition Codes:

```
        N <- 0; !condition codes cleared after BPT fault
        Z <- 0;
        V <- 0;
        C <- 0;
```

Exception:

none

Opcode:

```
  03    BPT     Breakpoint
```

Description:

In order to understand the operation of this instruction, read Chapter 5, Exceptions and Interrupts. This instruction is used, together with PSL<T>, to implement debugging facilities.

Notes:

1. Because the breakpoint exception is a fault, the PC and PSL pushed on the exception stack reflect the state of the processor at the beginning of the instruction, with PSL<TP> clear.

BUG Bugcheck

Format:

```
        opcode  {message.bx}
```

Operation:

```
        {fault to report error}
```

Condition Codes:

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

Exception:

reserved instruction

Opcode:

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. If the process is privileged to report bugs, a log entry is made. If the process is not privileged, a reserved instruction is signaled and is handled by the standard condition handler mechanism.

HALT   Halt

Format:

```
        opcode  {-(KSP).w*}
```

Operation:

```
        if PSL<VM> EQLU 1 and VMPSL<CUR_MOD> EQLU 0 then
                {VM-emulation trap};
                ! Exception frame contains no operands.

        if PSL<CUR_MOD> NEQU 0 then
                {privileged instruction fault}
                else
                {halt the processor};
```

Condition Codes:

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

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

Exceptions:

privileged instruction
VM emulation

Opcodes:

```
  00    HALT    Halt
```

Description:

In order to understand the operation of this instruction it is necessary to read Chapter 5, Exceptions and Interrupts. If the processor is executing a virtual machine and the virtual machine is in kernel mode, a VM-emulation trap occurs. 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;
```

Exception:

subscript range

Opcode:

```
  0A    INDEX   Compute 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 show some of the uses of INDEX.

3. The COBOL statements:

```
01  A-ARRAY.
    02  A PIC X(25) OCCURS 15 TIMES INDEXED BY I.
01  B PIC X(25).

MOVE A(I) TO B.
```

can be translated by a VAX COBOL compiler to:

```
INDEX   I(R11), #^X01, #^X0F, #^X19, #^X00, R0
MOVC3   #^X19, A-25(R11)[R0], B(R11)
```

4. The FORTRAN statements:

```
INTEGER*4       A(11:24), I
A(I) = 1
```

can be translated by a VAX FORTRAN compiler to:

```
INDEX   I(R11), #11, #24, #1, #0, R0
MOVL    #1, A-44(R11)[R0]
```

5. The Pascal statements:

```
     var
        i : integer;
        a : array[11..24] of integer;

        a[i] := 1
```

can be translated by a VAX Pascal compiler to:

```
INDEX   I,#11,#24,#1,#0,R0
MOVZBL  #1,A-44[R0]
```

MOVPSL Move from PSL

Format:

```
        opcode dst.wl
```

Operation:

```
        if PSL<VM> EQLU 0  then
                dst <- PSL
        else dst <- PSL<CM,TP> ' 0<29:28> ' PSL<FPD>
                    ' VMPSL<IS,CUR_MOD,PRV_MOD> ' 0<21>
                    ' VMPSL<IPL> ' 0<15:8>
                    ' PSL<DV,FU,IV,T,N,Z,V,C>;
```

Condition Codes:

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

Exceptions:

none

Opcode:

```
  DC    MOVPSL  Move from PSL
```

Description:

If the processor is not in VM mode, the destination operand is replaced by PSL. If the processor is in VM mode, the destination operand is replaced by the value of the virtual machine's PSL. Specifically, the values of the IS, CUR_MOD, PRV_MOD, and IPL fields are supplied from the VMPSL register; the values of the CM, TP, FPD, DV, FU, IV, T, N, Z, V, and C fields are supplied from the PSL register; and the values of the VM and the must-be-zero fields are zero.

NOP No Operation

Format:

```
        opcode
```

Operation:

```
        none
```

Condition Codes:

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

Exceptions:

```
        none
```

Opcode:

```
  01    NOP     No Operation
```

Description:

No operation is performed.

POPR Pop Registers

Format:

```
        opcode mask.rw, {(SP)+.r*}
```

Operation:

```
        tmp1 <- mask
        for tmp2 <- 0 step 1 until 14 do
        if tmp1<tmp2> EQL 1 then R[tmp2] <- (SP)+;
```

Condition Codes:

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

Exceptions:

none

Opcode:

```
  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, {-(SP).w*}
```

Operation:

```
tmp1 <- mask;
for tmp2 <- 14 step -1 until 0 do
if tmp1<tmp2> EQL 1 then -(SP) <- R[tmp2];
```

Condition Codes:

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

Exceptions:

none

Opcode:

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:

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

XFC Extended Function Call

Format:

```
        opcode  {unspecified operands}
```

Operation:

```
        {XFC fault};
```

Condition Codes:

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

Exceptions:

none

Opcode:

```
  FC    XFC     Extended Function Call
```

Description:

In order to understand the operation of this instruction, read Chapter 5. This instruction provides for user-defined extensions to the instruction set.

Change History:

Revision J. Rich Brunner, December 1989.

Revision H. Tim Leonard, May 1987.
- Support virtual machines.

Revision E. Al Thomas, September 1986.
- Add implied operands to instruction descriptions.
- Modify POPR operation to account for mask being located in a register.

Revision D. Tim Leonard, March 1985.
- Change the revision number to correspond to DEC Standard 032 rev number.
- Avoid references to particular versions of compilers in the notes to the INDEX instruction.
- Move bugcheck here, and delete the Chapter 4 section referring to instructions described in other Chapters. The index of instructions now does that.

Revision 7, repartition Chapter 4. Dileep Bhandarkar, 26 July 1982.

3.8 QUEUE INSTRUCTIONS

A queue is a circular, doubly linked list. A queue entry is specified by its address. The VAX architecture supports two distinct types of links: absolute and self-relative. An absolute link contains the absolute address of the entry to which it points. A self-relative link contains a displacement from the present queue entry. A queue is classified by the type of link it uses.

Because a queue contains redundant links, it is possible to create ill-formed queues. The VAX instructions produce UNPREDICTABLE results when used on ill-formed queues or on queues with overlapping entries.

3.8.1 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; it specifies the address of the succeeding queue entry. The second (highest addressed) longword is the backward link; it specifies the address of the preceding queue entry. A queue is specified by a queue header that 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. 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. Therefore, if more than one process can perform operations on a queue simultaneously, insertions and removals should only be done at the head or tail of the queue. If only one process (or one process at a time) can perform operations on a queue, insertions and removals can be made at other than the head or tail of the queue.

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.

3.8.2 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; it specifies the displacement of the succeeding queue entry from the present entry. The second longword (highest addressed) is the backward link; it specifies 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.

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. A hardware-supported, interlocked memory access mechanism is used to read the queue header. Bit <0> of the queue header is used as a secondary interlock and is set when the queue is being accessed.

If an interlocked queue instruction encounters the secondary interlock set then, in the presence of no exception conditions, 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.

If an interlocked queue instruction encounters both the secondary interlock set and an exception condition resulting from instruction execution then it is UNPREDICTABLE whether the exception will occur or if the instruction will terminate after setting the condition codes.

Chapter 5, INTERRUPTS AND EXCEPTIONS, discusses the results when multiple exceptions occur.

INSQHI Insert Entry into Queue at Head, Interlocked

Format:

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

Operation:

```
        ! Must have write access to header.
        ! Header must be quadword aligned.
        ! Header cannot be equal to entry.
        tmp1 <- (header){interlocked};  ! Acquire hardware interlock.
                                        ! tmp1<2:1> must be zero.
        if tmp1<0> EQLU 1 then
                begin
                (header){interlocked} <- tmp1;
                ! Release hardware lock.
                {set condition codes and terminate instruction};
                end;

        (header){interlocked} <- tmp1 OR 1;
        ! Release hardware lock,
        ! and set secondary interlock.

        If {all memory accesses can be completed} then
                ! Check if following addresses can be written
                ! without causing a memory management exception:
                !       entry
                !       header + tmp1
                ! Also, check for quadword alignment.
                begin
                tmp2 <- header - entry;
                (entry) <- tmp1 + tmp2; ! Forward link of entry.
                (entry + 4) <- tmp2;    ! Backward link of entry.
                (header + tmp1 + 4) <- -tmp1 - tmp2;
                ! Backward link of successor.
                {read (header){interlocked}};
                ! Acquire hardware interlock.
                (header){interlocked} <- -tmp2;
                ! Forward link of header, release interlocks.
                end;
        else
                begin
                {read (header){interlocked}};
                ! Acquire hardware interlock.
                (header){interlocked} <- tmp1;
                ! Release all interlocks.
                {backup instruction};
                {initiate fault};
                end;
```

Condition Codes:

```
        if {secondary interlock was clear} 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;
```

Exception:

reserved operand

Opcode:

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, the Z-bit 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, the queue is left in a consistent state (see Chapters 4 and 5). If the instruction fails to acquire the secondary interlock then, in the presence of no exception conditions, 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 4, 5, and 6).

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  ...             ; Attempt to insert entry.
        BEQL    1$              ; If queue was empty, branch.
        BCS     INSERT          ; If queue was locked, try again.
        CALL    WAIT(...)       ; If the entry was queued, wait.

1$:
```

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

5. The following are reserved operands: entry or header, when either is an address that is not quadword aligned (if its address bits<2:0> NEQU 0); (header), when (header)<2:1> is not zero; header and entry, when they are equal. In the last case, the queue is not altered.

6. If the instruction encounters both the secondary interlock set and an exception condition resulting from execution, then it is UNPREDICTABLE whether the exception will occur or if the instruction will terminate after setting the condition codes.

INSQTI Insert Entry into Queue at Tail, Interlocked

Format:

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

Operation:

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

        if tmp1<0> EQLU 1 then
        begin
        (header){interlocked} <- tmp1;           !release hardware interlock
        {set condition codes and terminate instruction};
        end;
        else
        begin
        (header){interlocked} <- tmp1 OR 1;     !set secondary interlock
                                                 !release hardware interlock
        If {all memory accesses can be completed} then
                !check if the following addresses can be written
                !without causing a memory management exception:
                !       entry
                !       header + (header + 4)
                !Also, check for quadword alignment
                begin
                tmp2 <- (header + 4);
                tmp3 <- header - entry;
                (entry) <- tmp3;                 !forward link of entry
                (entry + 4) <- tmp2 + tmp3;      !backward link of entry
                if {tmp2 NEQU 0} then (header+tmp2) <- -tmp3 - tmp2
                        else tmp1 <- -tmp3 - tmp2;
                                                 !forward link of predecessor
                (header+4) <- -tmp3;             !backward link of header
                {read (header){interlocked}};
                                                 !acquire hardware interlock
                (header){interlocked} <- tmp1;   !release interlocks
                end;
        else
                begin
                {read (header){interlocked}};
                                         !acquire hardware interlock
                (header){interlocked} <- tmp1;   !release interlocks
                {backup instruction};
                {initiate fault};
                end;
        end;
```

Condition Codes:

```
if {secondary interlock was clear} 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;
```

Exception:

reserved operand

Opcode:

5D INSQTI Insert Entry into Queue at Tail, Interlocked

Description:

The entry specified by the entry operand is inserted into the queue preceding the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise, the Z-bit 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, the queue is left in a consistent state (see Chapters 4 and 5). If the instruction fails to acquire the secondary interlock then, in the presence of no exception conditions, 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 4, 5, and 6).

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   ...              ; Attempt to insert entry.
        BEQL     1$               ; If queue was empty, branch.
        BCS      INSERT           ; If queue was locked, try again.
        CALL     WAIT(...)        ; If the entry was queued, wait.

1$:
```

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

5. The following are reserved operands: entry, header, or (header + 4), when anyone is an address that is not quadword aligned (if its address bits<2:0> NEQU 0); (header), when (header)<2:1> is not zero; header and entry, when they are equal. In the last case the queue is not altered.

6. If the instruction encounters both the secondary interlock set and an exception condition resulting from execution, then it is UNPREDICTABLE whether the exception will occur or if the instruction will terminate after setting the condition codes.

INSQUE Insert Entry in Queue

Format:

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

Operation:

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

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

Condition Codes:

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

Exceptions:

none

Opcode:

```
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, the Z-bit 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, the queue is left in a consistent state (see Chapters 4 and 5).

Notes:

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

   - Insert at head

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

   - Insert at tail

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

   - Insert after arbitrary predecessor

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

2. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (see Chapters 4, 5, and 6).

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

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

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

   1$:
   ```

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

REMQHI Remove Entry from Queue at Head, Interlocked

Format:

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

Operation:

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


        if tmp1<0> EQLU 1 then
                begin
                (header){interlocked} <- tmp1;
                                        !release hardware interlock
                {set condition codes and terminate instruction};
                end;
        else
                begin
                (header){interlocked} <- tmp1 OR 1;
                                        !set secondary interlock,
                                        !release hardware interlock
                If {all memory accesses can be completed} then
                        !check if the following can be done without
                        !causing a memory management exception:
                        !write addr operand
                        !read contents of header + tmp1 {if tmp1 NEQU 0}
                        !write into header + tmp1 + (header + tmp1)
                        !                                {if tmp1 NEQU 0}
                        !Also, check for quadword alignment
                        begin
                        tmp2 <- header + tmp1;
                        addr <- tmp2;
                        if {tmp1 EQL 0} then tmp3 <- header
                                else tmp3 <- tmp2 + (tmp2);
                        (tmp3 + 4) <- header - tmp3;
                                        !backward link of successor
                        {read (header){interlocked}};
                                        !acquire hardware interlock
                        (header){interlocked} <- tmp3 - header;
                                !forward link of header
                                !release all interlocks
                        end;
                else
                        begin
                        {read (header){interlocked}};
                                !acquire hardware interlock
                        (header){interlocked} <- tmp1;  !release interlocks
                        {backup instruction};
                        {initiate fault};
                        end;
        end;
```

Condition Codes:

```
        if {secondary interlock was clear} then
                begin
                N <- 0;
                Z <- (header) EQL 0;    !queue empty after removal
                V <- tmp1 EQL 0;        !no entry to remove
                C <- 0;
                end;
        else
                begin
                N <- 0;
                Z <- 0;
                V <- 1;             !did not remove anything
                C <- 1;             !secondary interlock failed
                end;
```

Exception:

reserved operand

Opcode:

5E REMQHI Remove Entry from Queue at Head, Interlocked

Description:

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

If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, the Z-bit 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, the queue is left in a consistent state (see Chapters 4 and 5). If the instruction fails to acquire the secondary interlock then, in the presence of no exception conditions, the instruction sets condition codes and terminates.

Notes:

1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (see Chapters 4, 5, and 6).

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 that cannot be completed results in a memory management exception even though the queue removal is not started.

6. The following are reserved operands: header or (header + (header)), when either is an address that is not quadword aligned (if its address bits<2:0> NEQU 0); (header), when (header)<2:1> is not zero; the header address operand and the address of the addr operand, when they are equal. In the last case, the queue is not altered.

7. If the instruction encounters both the secondary interlock set and an exception condition resulting from execution, then it is UNPREDICTABLE whether the exception will occur or if the instruction will terminate after setting the condition codes.

REMQTI Remove Entry from Queue at Tail, Interlocked

Format:

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

Operation:

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

if tmp1<0> EQLU 1 then
        begin
        (header){interlocked} <- tmp1;
                                !release hardware interlock
        {set condition codes and terminate instruction};
        end;
else
        begin
        (header){interlocked} <- tmp1 OR 1;
                                !set secondary interlock,
                                !release hardware interlock
        If {all memory accesses can be completed} then
                !check if the following can be done without
                !causing a memory management exception:
                !write addr operand
                !read contents of header + (header + 4)
                !                       {if tmp1 NEQU 0}
                !write into header + (header + 4)
                !  + (header + 4 + (header + 4)) {if tmp1 NEQU 0}
                !Also, check for quadword alignment
                begin
                addr <- header + (header + 4);
                tmp2 <- addr + (addr + 4);
                (header + 4) <- tmp2 - header;
                                !backward link of header
                tmp3 <- tmp1;   !save tmp1 to set Z correctly
                if {tmp2 EQL header} then tmp1 <- 0
                        else(tmp2) <- header - tmp2;
                            !forward link of predecessor
                {read (header){interlocked}};
                (header){interlocked} <- tmp1;  !release interlocks
                end;
        else
                begin
                {read (header){interlocked}};
                                !acquire hardware interlock
                (header){interlocked} <- tmp1;  !release interlocks
                {backup instruction};
                {initiate fault};
                end;
                end;
```

Condition Codes:

```
        if {secondary interlock was clear} then
                begin
                N <- 0;
                Z <- (header + 4) EQL 0;!queue empty after removal
                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;
```

Exception:

reserved operand

Opcode:

5F REMQTI Remove Entry from Queue at Tail, Interlocked

Description:

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

If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, the Z-bit 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, the queue is left in a consistent state (see Chapters 4 and 5). If the instruction fails to acquire the secondary interlock then, in the presence of no exception conditions, the instruction sets condition codes and terminates.

Notes:

1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (see Chapters 4, 5, and 6).

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

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

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

2$:
```

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

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

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

6. The following are reserved operands: header, (header + 4), or (header + (header + 4) + 4), when anyone is an address that is not quadword aligned (if its address bits<2:0> NEQU 0); (header), when (header)<2:1> is not zero; the header address operand and the address of the addr operand, when they are equal. In the last case, the queue is not altered.

7. If the instruction encounters both the secondary interlock set and an exception condition resulting from execution, then it is UNPREDICTABLE whether the exception will occur or if the instruction will terminate after setting the condition codes.

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

Exceptions:

none

Opcode:

0F REMQUE Remove Entry from Queue

Description:

The queue entry specified by the entry operand is removed from the queue. The address operand is replaced by the address of the entry removed. If there was no entry in the queue to be removed, the condition code V-bit is set; otherwise, it is cleared. If the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, the Z-bit 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, the queue is left in a consistent state (see Chapters 4 and 5).

Notes:

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

   o Remove at head

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

   o Remove at tail

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

   o 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 4, 5, and 6).

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 that cannot be completed results in a memory management exception even though the queue removal is not started.

Change History:

Revision J. Rich Brunner, December 1989
- o Clarify what happens when a secondary interlock can't be obtained in the presence of an exception.

Revision H. Tim Leonard, May 1987.

Revision E. Tim Leonard, September 1986.
- o Move the queue descriptions to Chapter 1.

Revision D. Tim Leonard, March 1985.
- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Fix simulation of condition codes in interlocked instructions, to use {secondary interlock was clear} rather than {insertion succeeded}

Revision 7, repartition Chapter 4. Dileep Bhandarkar, 26 July 1982.
- o Create separate file for this section.
- o Note that operations on ill-formed queues are UNPREDICTABLE.

3.9 FLOATING-POINT INSTRUCTIONS

NOTE

> H_floating instructions in the extended-accuracy group and floating-point instructions in the emulated-only group are optional in implementations of the VAX architecture. Execution of an omitted instruction results in a reserved instruction fault. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11, Implementation Options.

The floating-point instructions operate on four data types, termed F_floating, D_floating, G_floating, and H_floating. As noted above, implementations of the VAX architecture may exclude the H_floating data type.

3.9.1 Representation

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

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

where K is an integer and f is a non-negative fraction. For a non-vanishing number, K and f are uniquely determined by imposing the condition

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

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

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

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, 8 bits are reserved for the storage of the exponent K in excess 128 notation. Thus exponents from -128 to +127 could be represented, in biased form, by 0 to 255. For reasons given 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, 11 bits are reserved for the storage of the exponent in excess 1024 notation. Thus, exponents are restricted to -1023 to +1023 inclusive (in excess notation, 1 to 2047). In the H_floating data type 15 bits are reserved for the storage of the exponent in excess 16384 notation. Thus, exponents are restricted to -16383 to +16383 inclusive (in excess notation, 1 to 32767). A biased exponent of 0 is reserved for 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, VAX architecture 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 zeros, is treated as zero; this is the format generated by all floating-point instructions for which the result is zero.

A reserved operand is defined to be any bit pattern with a sign bit of 1 and a biased exponent of 0. In VAX architecture, all floating-point instructions generate a fault if a reserved operand is encountered. Scalar floating-point instructions (which are described in the following sections) never generate a reserved operand. However, vector floating-point instructions may generate reserved operands (see 13.8).

3.9.2 Overview of the Instruction Set

VAX architecture 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 following section on accuracy. The architecture 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 5.

VAX architecture also has a complete set of instructions for conversion from integer arithmetic types (byte, word, longword) to all floating types (F_floating, D_floating, G_floating, H_floating), and also for floating types to integer arithmetic types. VAX 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. A few instructions, however, may generate rounding error, floating overflow, or floating underflow, or may induce integer overflow. Details are given in the description of the CVT instructions.

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

All of the VAX floating-point instructions fault if a reserved operand is encountered. Floating-point instructions also fault on the occurrence of floating overflow or divide by zero. PSL<FU> is available to enable or disable an exception on underflow. If PSL<FU> is clear, no exception occurs on underflow and zero is returned as the result. If PSL<FU> 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 are completely discussed in Chapter 5.

3.9.3 Accuracy

General comments on the accuracy of the VAX 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 zeros, 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 the zero operand 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 the fractional factor is binary normalized with 53 or 113 bits for extended range double precision (G_floating), and extended range quadruple precision (H_floating), respectively.

\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 means 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 1, the rounded result is the chopped result incremented by an LSB (least significant bit).

2. If the rounding bit is 0, the rounded and chopped results are identical.

All VAX 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 0. Therefore, renormalization 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, its magnitude

   o Is always less than that of the true result for chopping

   o Is always less than that of the true result for rounding if the rounding bit is zero

   o Is greater than that of the true result for rounding if the rounding bit is one.

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

1. ADD or SUB:

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

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

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

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

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

    4. The constraint on no negations after the alignment shift may be replaced by keeping track of non-zero bits lost during the alignment shift, and then negating by one's complement if any "ones" were lost, and by two's complement if none were lost. If this is done, the error bound will be (1/2) LSB.

2. MUL:

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

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

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

3. DIV:

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

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

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

3.9.4 Programming Considerations

In order to be consistent with the floating-point instruction set which faults on reserved operands, software-implemented floating-point functions (the absolute function, for example) 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 content of register Rn is used as both a part of a floating-point input operand (operand type .rf, .rd, .rg, .rh, .mf, .md, .mg, or .mh) and as an address in an addressing mode that modifies Rn (autoincrement, autodecrement, or autoincrement deferred), the value of the floating-point operand is UNPREDICTABLE.

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 <- 0;
        C <- 0;
```

Exceptions:

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

Opcodes:

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

Description:

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

Notes:

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

2. On floating underflow, a fault occurs if PSL<FU> is set. Zero is stored as the result of floating underflow only if PSL<FU> is clear. On a floating underflow fault, the sum operand is unaffected. If PSL<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.

4. The ADDHx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

CLR Clear

Format:

```
opcode dst.wx
```

Operation:

```
dst <- 0;
```

Condition Codes:

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

Exceptions:

none

Opcodes:

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

Description:

The destination operand is replaced by 0.

Notes:

1. "CLRx dst" is equivalent to "MOVx #0, dst", but is 5 (Ffloating), or 9 (Dfloating or Gfloating), or 17 (Hfloating) bytes shorter.

2. The CLRH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

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;
```

Exception:

reserved operand

Opcodes:

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

Description:

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

Notes:

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

2. The CMPH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

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
floating overflow
floating underflow
reserved operand

Opcodes:

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

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

Description:

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

CVTBF	exact
CVTBD	exact
CVTBG	exact
CVTBH	exact
CVTWF	exact
CVTWD	exact
CVTWG	exact
CVTWH	exact
CVTLF	rounded
CVTLD	exact
CVTLG	exact
CVTLH	exact

CVTFB	truncated
CVTDB	truncated
CVTGB	truncated
CVTHB	truncated
CVTFW	truncated
CVTDW	truncated
CVTGW	truncated
CVTHW	truncated
CVTFL	truncated
CVTRFL	rounded
CVTDL	truncated
CVTRDL	rounded
CVTGL	truncated
CVTRGL	rounded
CVTHL	truncated
CVTRHL	rounded

CVTFD	exact
CVTFG	exact
CVTFH	exact
CVTDF	rounded
CVTDH	exact
CVTGF	rounded
CVTGH	exact
CVTHF	rounded
CVTHD	rounded
CVTHG	rounded

Notes:

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

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

5. When CVTRFL, CVTRDL, CVTRGL, and CVTRHL round, the rounding is done in sign magnitude, before conversion to two's complement.

6. The CVTxH, CVTHx, and CVTRHL instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

DIV Divide

Format:

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

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

Operation:

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

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

Condition Codes:

```
N <- quo LSS 0;
Z <- quo EQL 0;
V <- 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, a fault occurs if PSL<FU> is set. Zero is stored as the result of floating underflow only if PSL<FU> is clear. On a floating underflow fault, the quotient operand is unaffected. If PSL<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 item 3 above.

5. The DIVHx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

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; 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, a fault occurs if PSL<FU> is set. The integer and fraction parts are replaced by 0 on the occurrence of floating underflow only if PSL<FU> is clear. On a floating underflow fault, the integer and fraction parts are unaffected. If PSL<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. Integer overflow is possible, however, in the absence of floating overflow.

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

6. The integer and fraction are separated before the integer is converted to two's complement and before the fraction is rounded. Because rounding occurs after separation, the value of the fraction operand may be 1.

7. The EMODx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

8. The EMODx instructions are in the emulated-only group and are unlikely to be implemented in future VAX processors. Software developers are advised to avoid using these instructions.

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;
```

Exception:

reserved operand

Opcodes:

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

Description:

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

Notes:

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

2. The MNEGH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

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;
```

Exception:

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:

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

2. The MOVH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

MUL Multiply

Format:

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

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

Operation:

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

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

Condition Codes:

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

Exceptions:

floating overflow
floating underflow
reserved operand

Opcodes:

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

Description:

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

Notes:

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

2. On floating underflow, a fault occurs if PSL<FU> is set. Zero is stored as the result of floating underflow only if

PSL<FU> is clear. On a floating underflow fault, the product operand is unaffected. If PSL<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.

4. The MULHx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

POLY Polynomial Evaluation

Format:

```
opcode arg.rf, degree.rw, tbladdr.ab, {R0-3.wl}  !POLYF

opcode arg.rd, degree.rw, tbladdr.ab, {R0-5.wl}  !POLYD

opcode arg.rg, degree.rw, tbladdr.ab, {R0-5.wl}  !POLYG

opcode arg.rh, degree.rw, tbladdr.ab,
       {R0-5.wl,-16(SP):-1(SP).wb}               !POLYH
```

Operation:

```
tmp1 <- degree;
if tmp1 GTRU 31 then {initiate reserved operand fault};
tmp2 <- tbladdr;
tmp3 <- {(tmp2)+};        !tmp3 accumulates the partial result
                          !tmp3 is of type x
if POLYH then -(SP) <- arg;
while tmp1 GTRU 0 do
begin                     !Begin computation loop
tmp4 <- {arg * tmp3};     !tmp4 accumulates new partial result.
                          !tmp3 has old partial result.

        ! Perform multiply, and retain either 31 or 32 (POLYF),
        ! either 63 or 64 (for POLYD, POLYG),  and either  127
        ! or 128 (for POLYH) most  significant  bits  of  the
        ! fraction  by  truncating  the  unnormalized  product
        ! toward zero.  (The  most  significant  bits  in  the
        ! product fraction will be 01 (binary) if the  product
        ! fraction  is LSS 1/2 and  GEQ  1/4,  and  the  most
        ! significant bit  will  be  1  otherwise.)   Use  the
        ! result in the following add operation.

tmp4 <- tmp4 + (tmp2);
        ! Align fractions, perform add, and retain either  the
        ! 31 or 32 (for POLYF), 63 or 64 (for POLYD or POLYG),
        ! and  127 or 128 (for POLYH) most significant bits of
        ! the fraction by truncating  the  unnormalized result
        ! toward zero.  Normalize, and round to type x.  Check
        ! for  overflow  and underflow only after the combined
        ! multiply, add, normalize, round sequence.

if OVERFLOW then FLOATING OVERFLOW FAULT
if UNDERFLOW then
        begin
        if PSL<FU> EQL 1 then FLOATING UNDERFLOW FAULT;
        tmp4 <- 0;      !force result to 0;
        end;
tmp1 <- tmp1 - 1;
tmp2 <- tmp2 + {size of data type};
```

```
        tmp3 <- tmp4;   !update partial result in tmp3
        end;
        if POLYF then
                begin
                R0 <- tmp3;
                R1 <- 0;
                R2 <- 0;
                R3 <- tmp2;
                end;
        if POLYD or POLYG then
                begin
                R1'R0 <- tmp3;
                R2 <- 0;
                R3 <- tmp2;
                R4 <- 0;
                R5 <- 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 <- 0;
        C <- 0;
```

Exceptions:

        floating overflow
        floating underflow
        reserved operand

Opcodes:

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

Description:

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

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

where x is the argument and d is the degree. 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.

```
 31                         16 15 14           7 6          0
+----------------------------+--+---------------+------------+
|          fraction          | S|    exponent   |  fraction  | :R0
+----------------------------+--+---------------+------------+
|                            0                               | :R1
+------------------------------------------------------------+
|                            0                               | :R2
+------------------------------------------------------------+
|               table address + degree*4 + 4                 | :R3
+------------------------------------------------------------+
```

Figure 3-3 Result Registers After POLYF

```
 31                         16 15 14           7 6          0
+----------------------------+--+---------------+------------+
|          fraction          | S|    exponent   |  fraction  | :R0
+----------------------------+--+---------------+------------+
|          fraction          |          fraction             | :R1
+----------------------------+-------------------------------+
|                            0                               | :R2
+------------------------------------------------------------+
|               table address + degree*8 + 8                 | :R3
+------------------------------------------------------------+
|                            0                               | :R4
+------------------------------------------------------------+
|                            0                               | :R5
+------------------------------------------------------------+
```

Figure 3-4 Result Registers After POLYD

```
 31                         16 15 14                4 3     0
+----------------------------+--+--------------------+-------+
|          fraction          | S|      exponent      |  fra  | :R0
+----------------------------+--+--------------------+-------+
|          fraction          |          fraction             | :R1
+----------------------------+-------------------------------+
|                            0                               | :R2
+------------------------------------------------------------+
|               table address + degree*8 + 8                 | :R3
+------------------------------------------------------------+
|                            0                               | :R4
+------------------------------------------------------------+
|                            0                               | :R5
+------------------------------------------------------------+
```

Figure 3-5 Result Registers After POLYG

```
 31                                 16 15 14                              0
+-----------------------------------+--+-----------------------------------+
|             fraction              | S|           exponent                | :R0
+-----------------------------------+--+-----------------------------------+
|             fraction              |              fraction                | :R1
+-----------------------------------+--------------------------------------+
|             fraction              |              fraction                | :R2
+-----------------------------------+--------------------------------------+
|             fraction              |              fraction                | :R3
+-----------------------------------+--------------------------------------+
|                                   0                                      | :R4
+--------------------------------------------------------------------------+
|                table address + degree*16 + 16                            | :R5
+--------------------------------------------------------------------------+
```

Figure 3-6 Result Registers After POLYH

Notes:

1. After execution, the registers are as shown in Figures 3-3 through 3-6.

2. On a floating fault:

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

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

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

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

   Arg is saved on the stack in use during the faulting instruction. \Arg appears on the stack as it would if it were saved by a MOVH arg, -(SP)\ Implementation-dependent 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]. If the degree is 0 and either the argument or C[0] is a reserved operand, a reserved operand fault occurs.

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

5. On a reserved operand fault:

   - If PSL<FPD> = 0, the reserved operand is either the degree operand (greater than 31), or the argument operand, or some coefficient.

   - If PSL<FPD> = 1, the reserved operand is a coefficient, and R3 (except for POLYH) or R5 (for POLYH) is pointing at the value that caused the exception.

   - The state of the saved condition codes and the other registers is UNPREDICTABLE. If the reserved operand is changed and the contents of the condition codes and all registers are preserved, the fault is continuable.

6. On floating underflow after the rounding operation at any iteration of the computation loop, a fault occurs if PSL<FU> is set. If PSL<FU> is clear, the temporary result (tmp3) is replaced by 0 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, the result is C[0]. Additionally, if one of the coefficients in the table (other than C[0]) is a reserved operand, whether a reserved operand fault occurs is UNPREDICTABLE.

9. The POLYx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

10. The POLYx instructions are in the emulated-only group and are unlikely to be implemented in future VAX processors. Software developers are advised to avoid using these instructions.

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

Example:

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

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

USAGE NOTE

The MicroVAX-II, VAX 8200, and VAX 8300 processors have been granted a waiver for a bug in their implementations of POLY. These processors truncate the sum after normalization, not before normalization. If the ADD step produces a large cancellation, the low order bits of the sum can be shifted by normalization until they appear in the sum.

If POLY is used to evaluate polynomials whose successive terms decrease by a factor of at least 1/4, as the terms are traversed from lowest towards highest degree, then this bug will not affect the results. The math-library routines sold by DIGITAL use polynomials which meet this criterion. If customer-written software uses the POLY instruction with polynomials that do not meet this criterion, the results of the instruction may differ from those produced by other VAX processors.

Example:

```
TEST :   POLYD    ARG,    #1,  TABLE      ; Evaluate the
                                          ; polynomial

ARG  :   .QUAD    XFFFFDFFFFFFF407F       ; Argument

TABLE :  .QUAD    X0005E000FFFFDFFF       ; Coefficient 1.
         .QUAD    X0001000000006000       ; Coefficient 0.

; The correct answer is        ^X0000E800FFFF52FF

; The Microvax-II, VAX 8200,
; and VAX 8300 answer is       ^X0000E808FFFF52FF
```

IMPLEMENTATION NOTE

\Each multiplication step retains an unnormalized result of 31 or 32, 63 or 64, and 127 or 128 bits, for

POLYF, POLYD, POLYG, or POLYH, respectively. This result consists of two parts: a mantissa of 24, 56, 53, or 113 high-order bits; and 7 or 8, 7 or 8, 10 or 11, and 14 or 15 low-order guard bits. Although the precision provided may depend on the processor type, the datatype, and whether or not an FPA or an FPU is installed, the precision provided must not depend on the operand values. That is, POLYF on a 780 with the FPA turned on must use either 31 or 32 bits of precision all the time, not 31 bits sometimes and 32 bits other times.\

IMPLEMENTATION NOTE

\The following special case has caused past bugs:

1. The guard bits equal exactly 1/2 the fraction LSB (for example, 1000000 binary for POLYF), and

2. The next coefficient to add is negative.

When this occurs, the coefficient must not be approximated by zero even if it is very small, say of magnitude epsilon. A zero approximation will produce an incorrect extra round in this case. 1/2*LSB-epsilon must not cause rounding.\

```
        .ENTRY  START,0
        POLYF   ARG,DEGR,TABL
        CMPL    #^XFE8A766C, R0          ; Check for correct answer.
        RET
ARG::   .LONG   ^X00004040
DEGR::  .LONG   ^X1
TABL::  .LONG   ^XFF07769D
        .LONG   ^X01018A01
        .END    START
```

SUB Subtract

Format:

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

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

Operation:

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

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

Condition Codes:

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

Exceptions:

floating overflow
floating underflow
reserved operand

Opcodes:

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

Description:

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

Notes:

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

2. On floating underflow, a fault occurs if PSL<FU> is set. Zero is stored as the result of floating underflow only if PSL<FU> is clear. On a floating underflow fault, the

difference operand is unaffected. If PSL<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.

4. The SUBHx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

TST Test

Format:

```
        opcode src.rx
```

Operation:

```
        src - 0;
```

Condition Codes:

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

Exception:

        reserved operand

Opcodes:

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

Description:

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

Notes:

1. "TSTx src" is equivalent to "CMPx src, #0", but is 5 (Ffloating) or 9 (Dfloating or Gfloating) or 17 (Hfloating) bytes shorter.

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

3. The TSTH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

Change History:

Revision J. Rich Brunner, December 1989

- o Vector instructions can generate floating-point reserved operands.

Revision H. Tim Leonard, May 1987.

Revision F. Al Thomas, November 1986.

- o Add new instruction-implementation rules.

Revision E. Al Thomas, September 1986.

- o Add clarification to notes 5 and 6 of EMOD.
- o Add implied operands to POLYx description.
- o Include all the floating point instructions as optional.
- o Describe the EMOD/POLY subset.

Revision D1. Tim Leonard, January 1986.

- o Describe POLY waiver for MicroVAX and V-11.
- o POLY may have more accuracy, depending on implementation.

Revision D. Tim Leonard, March 1985.

- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Add note on rounding in CVTRxL and EMODx instructions.
- o Floating point data types and instructions may be omitted, and emulation may use stack space.
- o V bit is not set on floating overflow since there are no traps.

Revision 7. Dileep Bhandarkar, 26 July 1982.

- o Reserved operand fault on argument even if degree is 0 (POLY).
- o Remove specification of floating traps. All VAX-11/780s will be retrofitted for floating faults.
- o Clarify truncation for ADD operation of POLY.

Revision 6, clarify POLY. Dileep Bhandarkar, 27 March 1980.
Revision 5, add G_floating and H_floating. Dileep Bhandarkar, 31 January 1979.

- o Add MOVG, MOVH, CLRH, CLRO, MNEGG, MNEGH, CVTBG, CVTBH, CVTWG, CVTWH, CVTLG, CVTLH, CVTGB, CVTHB, CVTGW, CVTHW, CVTGL, CVTRGL, CVTHL, CVTRHL, CVTGH, CVTHG, CVTGF, CVTHF, CVTFG, CVTFH, CVTHD, CVTDH, CMPG, CMPH, TSTG, TSTH, ADDG2, ADDG3, ADDH2, ADDH3, SUBG2, SUBG3, SUBH2, SUBH3, MULG2, MULG3, MULH2, MULH3, DIVG2, DIVG3, DIVH2, DIVH3, EMODG, EMODH, POLYG, POLYH.
- o Change floating underflow and overflow exceptions to faults.

meeting. Bill Strecker, 10 June 1976. Revision 4, ECO to POLY and ECO to facilitate high-speed FP. Bill Strecker, 24 March 1977.

- o Remove destination operand specifier for POLY.
- o Reverse the the order of coefficients for POLY.

- POLYF will not set R4 or R5.
- Change the name of second operand of POLY to degree.
- For POLY give reserved operand fault if degree operand > 31.
- Define certain pathological addressing combinations to give UNPREDICTABLE results to facilitate high speed floating point implementations.
- Add introduction, rounding and accuracy.
- POLYx UNPREDICTABLE whether reserved operand if arg = 0.
- POLYx coefficient reserved can give FPD=0 fault.

Revision 3, ECOs 12 through 18, results of April Task Force review and May 25

- Reserved operand aborts become faults.
- Specify 0 divisor behavior for DIVF, DIVD.
- Change pointer to longword or address; make it 32 bits.
- Add MINU function in ISP.
- Explicitly give SEXT or ZEXT in all cases needed.
- MOVF, MOVD take reserved operand fault.
- Remove round bit.
- Floating overflow, underflow get reserved, 0 respectively.
- Specified condition codes on all exceptions.
- Remove CHOPF, CHOPD.
- Update EMODF, EMODD per ECO 18.
- Change EMODD to produce longword fraction.
- Split into separate specifications.

Revision 2, ECOs 1 through 11. Bill Strecker, 16 March 1976.
Revision 1, initial distribution. Strecker, 25 September 1975.

3.10 CHARACTER-STRING INSTRUCTIONS

NOTE

> The MATCHC, MOVTC and MOVTUC instructions are optional in implementations of the VAX architecture. Execution of an omitted instruction results in an emulated instruction exception. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11, Implementation Options.

A character string is specified by two operands:

- An unsigned word operand that specifies the length of the character string in bytes
- 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 that maintains updated addresses and state during the execution of the instruction. When instruction execution is completed, these registers are available to software to use as string specification operands for a subsequent instruction on a contiguous character string. During the execution of the instructions, pending interrupt conditions are tested. If any is found, the control block is updated, a first-part-done bit is set in the PSL, and the instruction interrupted (see Chapter 5). After the interruption, the instruction resumes transparently.

The format of the control block is shown in Figure 3-7. 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.

```
 31                            16 15                             0
+-------------------------------+--------------------------------+
|                               |           length 1             | :R0
+-------------------------------+--------------------------------+
|                           address 1                            | :R1
+-------------------------------+--------------------------------+
|                               |           length 2             | :R2
+-------------------------------+--------------------------------+
|                           address 2                            | :R3
+-------------------------------+--------------------------------+
|                               |           length 3             | :R4
+-------------------------------+--------------------------------+
|                           address 3                            | :R5
+----------------------------------------------------------------+
```

Figure 3-7 Character-String-Instruction Control Block

CMPC Compare Characters

Format:

```
        opcode len.rw, src1addr.ab, src2addr.ab,   3 operand
               {R0-3.wl}

        opcode src1len.rw, src1addr.ab, fill.rb,
               src2len.rw, src2addr.ab, {R0-3.wl}  5 operand
```

Operation:

```
   ! In all cases and flows, tmpa and tmpb hold the last two bytes that
   ! were compared and so are used to set the condition codes at the end.
```

CMPC3 flow:

```
        tmp1 <- len;       tmp2 <- src1addr;                 !3 operand
        tmp4 <- src2addr;
        tmpa <- 0;         tmpb <- 0;                        !init compare regs

        while {tmp1 NEQU 0} do                               !SRC1 & SRC2
              begin                                          !not zero-length
              tmpa <- (tmp2);  tmpb <- (tmp4);
              if tmpa EQL tmpb then                          !bytes match
                 begin
                 tmp1 <- tmp1 - 1; tmp2 <- tmp2 + 1;
                 tmp4 <- tmp4 + 1;
                 end
              else [exit while loop];                        !mismatch
              end;

        tmp3 <- tmp1;
        [exit to Register Save flow];
```

CMPC5 flow:

```
        tmp1 <- src1len;  tmp2 <- src1addr;                  !5 operand
        tmp3 <- src2len;  tmp4 <- src2addr;
        tmpa <- 0;        tmpb <- 0;                         !init compare regs

        while {tmp1 NEQU 0} AND {tmp3 NEQU 0} do             !SRC1 & SRC2
              begin                                          !not zero-length
              tmpa <- (tmp2);  tmpb <- (tmp4);
              if tmpa EQL tmpb then                          !bytes match
                 begin
                 tmp1 <- tmp1 - 1;  tmp2 <- tmp2 + 1;
                 tmp3 <- tmp3 - 1;  tmp4 <- tmp4 + 1;
                 end
              else [exit while loop];                        !mismatch
              end;
        while ({tmp1 NEQU 0} AND {tmp3 EQU 0}) do         !SRC2 zero-length
              begin                                        !or exhausted, SRC1
              tmpb <- fill;                                !has bytes left
              tmpa <- (tmp2);
              if tmpa EQL fill then                        !bytes match
                 begin
                 tmp1 <- tmp1 - 1;  tmp2 <- tmp2 + 1;
                 end
              else [exit while loop];                      !mismatch
              end;

        while ({tmp3 NEQU 0} AND {tmp1 EQU 0}) do         !SRC1 zero-length
              begin                                        !or exhausted, SRC2
              tmpa <- fill;                                !has bytes left
              tmpb <- (tmp4);
              if fill EQL tmpb then                        !bytes match
                 begin
                 tmp3 <- tmp3 - 1;  tmp4 <- tmp4 + 1;
                 end
              else [exit while loop];                      !mismatch
              end;
```

Register Save flow:

```
        r0 <- tmp1;
        r1 <- tmp2;
        r3 <- tmp3;
        r4 <- tmp4;
```

Condition codes:

```
        N <- tmpa LSS tmpb;
        Z <- tmpa EQL tmpb;
        V <- 0;
        C <- tmpa LSSU tmpb;
```

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 (PSL<Z> is set and PSL<N>, PSL<V>, and PSL<C> are cleared.)

Notes:

1. After execution of CMPC3:

   R0 = number of bytes remaining in string 1 (including byte that terminated comparison); R0 is zero only if strings are equal

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

   R2 = R0

   R3 = address of the byte in string 2 that 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 that 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 that 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 that 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 that 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, PSL<Z> is set and PSL<N>, PSL<V>, and PSL<C> are cleared just as in the case of two equal strings.

LOCC   Locate Character

Format:

```
        opcode char.rb, len.rw, addr.ab, {R0-1.wl}
```

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcode:

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, {R0-3.wl}
```

Operation:

```
        tmp1 <- objlen;
        tmp2 <- objaddr;
        tmp3 <- srclen;
        tmp4 <- srcaddr;
        tmp5 <- tmp1;

        while {tmp1 NEQU 0} AND {tmp3 GEQU tmp1} do
                begin
                if (tmp2) EQL (tmp4) then
                        begin
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 + 1;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 + 1;
                        end
                else
                        begin
                        tmp2 <- tmp2 - ZEXT (tmp5-tmp1);
                        tmp3 <- {tmp3 - 1} + {tmp5-tmp1};
                        tmp4 <- {tmp4 + 1} - ZEXT (tmp5-tmp1);
                        tmp1 <- tmp5;
                        end;
                end;

        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

Opcode:

39 MATCHC Match Characters

Description:

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

Notes:

1. After execution:

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

   R1 = if a match occurred, the address of one byte beyond the object string (that is, objaddr + objlen); otherwise, the address of the object string

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

   R3 = if a match occurred, the address of one byte beyond the last byte matched; otherwise, the address of one byte beyond the source string (that is, srcaddr + srclen).

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

2. If both strings have zero length or if the object string has zero length, PSL<Z>is set and registers R0 through 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, PSL<Z> is cleared and registers R0 through R3 are left just as though the substring were not found.

4. The MATCHC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

5. The MATCHC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

MOVC   Move Character

Format:

```
        opcode len.rw, srcaddr.ab, dstaddr.ab,        3 operand
               {R0-5.wl}

        opcode srclen.rw, srcaddr.ab, fill.rb,
               dstlen.rw, dstaddr.ab, {R0-5.wl}       5 operand
```

Operation:

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

```
tmp1 <- srclen;                            !5 operand
tmp2 <- srcaddr;
tmp3 <- dstlen;
tmp4 <- dstaddr;
if tmp2 GTRU tmp4 then
        begin
        while {tmp1 NEQU 0} AND {tmp3 NEQU 0} do
                begin
                (tmp4) <- (tmp2);
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 + 1;
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 + 1;
                end;
        while tmp3 NEQU 0 do
                begin
                (tmp4) <- fill;
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 + 1;
                end;
        R1 <- tmp2;
        R3 <- tmp4;
        end
else
        begin
        tmp5 <- MINU(tmp1, tmp3);
        tmp6 <- tmp3;
        tmp2 <- tmp2 + ZEXT(tmp5);
        tmp4 <- tmp4 + ZEXT(tmp6);
        while tmp3 GTRU tmp1 do
                begin
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 - 1;
                (tmp4) <- fill;
                end;
        while tmp3 NEQU  0 do
                begin
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 - 1;
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 - 1;
                (tmp4) <- (tmp2);
                end;
        R1 <- tmp2 + ZEXT (tmp5);
        R3 <- tmp4 + ZEXT (tmp6);
        end;
R0 <- tmp1;
R2 <- 0;
R4 <- 0;
R5 <- 0;
```

Condition Codes:

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

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

Exceptions:

none

Opcodes:

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

Description:

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

Notes:

1. After execution of MOVC3:

   R0 = 0

   R1 = address of one byte beyond the source string

   R2 = 0

   R3 = address of one byte beyond the destination string.

   R4 = 0

   R5 = 0.

2. After execution of MOVC5:

   R0 = number of unmoved bytes remaining in source string; R0 is non-zero only if source string is longer than destination string

   R1 = address of one byte beyond the last byte in source string that was moved

   R2 = 0

   R3 = address of onе byte beyond the destination string

   R4 = 0

   R5 = 0.

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

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

MOVTC Move Translated Characters

Format:

```
        opcode srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab,
               dstlen.rw, dstaddr.ab, {R0-5.wl}
```

Operation:

```
        tmp1 <- srclen;
        tmp2 <- srcaddr;
        tmp3 <- dstlen;
        tmp4 <- dstaddr;
        if tmp2 GTRU tmp4 then
                begin
                while {tmp1 NEQU 0} AND {tmp3 NEQU 0}
                        begin
                        (tmp4) <- (tbladdr + ZEXT((tmp2)));
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 + 1;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 + 1;
                        end;
                while {tmp3 NEQU 0} do
                        begin
                        (tmp4) <- fill;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 + 1;
                        end;
                R1 <- tmp2;
                R5 <- tmp4;
                end;
        else
                begin
                tmp5 <- MINU(tmp1,tmp3);
                tmp6 <- tmp3;
                tmp2 <- tmp2 + ZEXT(tmp5);
                tmp4 <- tmp4 + ZEXT(tmp6);
                while tmp3 GTRU tmp1 do
                        begin
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 - 1;
                        (tmp4) <- fill;
                        end;
                while tmp3 NEQU 0 do
                        begin
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 - 1;
                        tmp3 <- tmp3 - 1;
                        tmp4 <- tmp4 - 1;
                        (tmp4) <- (tbladdr + ZEXT((tmp2)));
                        end;
                R1 <- tmp2 + ZEXT(tmp5);
                R5 <- tmp4 + ZEXT(tmp6);
                end;
        R0 <- tmp1;
        R2 <- 0;
        R3 <- tbladdr;
        R4 <- 0;
```

Condition Codes:

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

Exceptions:

none

Opcode:

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:

1. 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.

2. The MOVTC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

3. The MOVTC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

MOVTUC Move Translated Until Character

Format:

```
        opcode srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw,
               dstaddr.ab, {R0-5.wl}
```

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

Opcode:

```
   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:

1.

After execution:

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

R1 = address of the byte that resulted in destination string exhaustion or escape; or if no exhaustion or escape, address of 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 that would have received the translated byte that caused the escape or would have received a translated byte if the source string were not exhausted; or if no exhaustion or escape, the address of one byte beyond the destination string.

2. The MOVTUC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

3. The MOVTUC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

SCANC Scan Characters

Format:

```
        opcode len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.wl}
```

Operation:

```
        tmp1 <- len;
        tmp2 <- addr;
        if tmp1 GTRU 0 then
                begin
                while {tmp1 NEQU 0} AND
                {{(tbladdr + ZEXT((tmp2))) AND mask} EQL 0} do
                        begin
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 + 1;
                        end;
                end;
        R0 <- tmp1;
        R1 <- tmp2;
        R2 <- 0;
        R3 <- tbladdr;
```

Condition Codes:

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

Exceptions:

none

Opcode:

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 that produced the non-zero AND result); R0 is zero only if there was no non-zero AND result

   R1 = address of the byte that 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, PSL<Z> is set just as though the entire string were scanned.

SKPC Skip Character

Format:

```
        opcode  char.rb, len.rw, addr.ab, {R0-1.wl}
```

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcode:

```
  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, PSL<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, {R0-3.wl}
```

Operation:

```
        tmp1 <- len;
        tmp2 <- addr;
        if tmp1 GTRU 0 then
                begin
                while {tmp1 NEQU 0} AND
                        {{(tbladdr + ZEXT((tmp2))) AND mask} NEQ 0} do
                        begin
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 + 1;
                        end;
                end;
        R0 <- tmp1;
        R1 <- tmp2;
        R2 <- 0;
        R3 <- tbladdr;
```

Condition Codes:

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

Exceptions:

none

Opcode:

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 that produced the zero AND result); R0 is zero only if there was no zero AND result

   R1 = address of the byte that produced a zero AND result; 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 PSL<Z> is set just as though the entire string were spanned.

Change History:

Revision J. Rich Brunner, December 1989.
- o Revised CMPCx pseudo-code for correctness and clarity.

Revision H. Tim Leonard, May 1987.

Revision F. Al Thomas, November 1986.
- o Add new instruction-implementation rules.

Revision E. Al Thomas, September 1986.
- o Add implied operands to instruction descriptions.
- o Include CMPC, LOCC, MATCHC, MOVTC, MOVTUC, SCANC, SKPC, and SPANC as optional instructions.

Revision D. Tim Leonard, March 1985.
- o Note that string instructions may be omitted, and emulation may use the stack.
- o Change the revision number to correspond to DEC Standard 032 rev number.

Revision 7, repartition Chapter 4. Dileep Bhandarkar, 26 July 1982.
Revision 6. Dileep Bhandarkar, 26 January 1980.
- o Correct MATCHC description.
- o MOVTUC allows in place translation.
- o Change CRC to clear C bit.

Revision 5, MATCHC ECO. Dileep Bhandarkar, 26 October 1978.
Revision 4. Bill Strecker, 31 March 1977.
- o ECO to complete documentation of CRC Instruction.
- o Clarify null strings.
- o Add AUTODIN-II CRC.

Revision 3, ECOs 12 through 18, results of the April Task Force review and the May 25 meeting. Bill Strecker, 10 June 1976.
- o Reserved operand aborts become faults.
- o Add SKPC.
- o Change pointer to longword or address; make it 32 bits.
- o Add MINU function in ISP.
- o Explicitly give SEXT or ZEXT in all cases needed.
- o Specified condition codes on all exceptions.
- o MOVTC does not translate fill.
- o Increase registers used by MOVC3 and MOVC5 to 6.
- o Change condition codes setting MOVC, MOVTC, MOVTUC.
- o Add MOVTUC, MATCHC.
- o Add CRC per ECO 12.
- o Change CRC table operand from .al to .ab.
- o Split into separate specifications.

Revision 2, ECOs 1 through 11. Bill Strecker, 16 March 1976.
Revision 1, initial distribution. Bill Strecker, 25 September 1975.

3.11 CYCLIC-REDUNDANCY-CHECK INSTRUCTION

NOTE

> The cyclic-redundancy-check instruction is optional in implementations of the VAX architecture. Execution of the instruction, when omitted, results in an emulated instruction exception. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11, Implementation Options.

This instruction is designed to implement the calculation and checking of a cyclic redundancy check (CRC) for any CRC polynomial up to 32 bits. Cyclic redundancy checking is an error-detection method involving a division of the data stream by a CRC polynomial. The data stream is represented as a standard VAX string in memory. Error detection is accomplished by computing the CRC at the source and again at the destination, comparing the CRC computed at each end. The choice of the polynomial is such as to minimize the number of undetected block errors of specific lengths. For information about choosing CRC polynomials, see the article written by A. Marton and T. Frambs, "A Cyclic Redundancy Checking Algorithm," Honeywell Computer Journal, No. 3, 1971, pp 140-142.

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 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 he different if the data stream were 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 1, 2, or 4 bits at a time using the appropriate entries in a specially constructed table. The instruction produces a 32-bit CRC. For shorter polynomials, the result must be extracted from the 32-bit field. The data stream must be a multiple of 8 bits in length. If it is not, the stream must be right-adjusted in the string with leading 0 bits.

CRC Calculate Cyclic Redundancy Check

Format:

```
        opcode  tbl.ab, inicrc.rl, strlen.rw, stream.ab, {R0-3.wl}
```

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 notes for limit,s,i
                        tmp3 <- ZEXT(tmp3<31:s>) XOR
                                (tmp4 + {4*ZEXT(tmp3<s-1:0>*i)};
                tmp1 <- tmp1 -1;
                end;
        R0 <- tmp3;
        R1 <- 0;
        R2 <- 0;
        R3 <- tmp2;
```

Condition Codes:

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

Exceptions:

none

Opcode:

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 0 fill.

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

3. The following algorithm can be used to calculate the CRC table given a polynomial expressed as follows:

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

   This routine is available as system library routine LIB$CRC_TABLE (poly.rl, table.ab). The bits of the poly operand, taken right to left, represent the coefficients of the polynomial, taken left to right and skipping the most significant bit. The table is the location of a 64-byte (16-longword) table into which the result will be written.

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

4. Table 3-1 describes some commonly used CRC polynomials.

5. This instruction produces an UNPREDICTABLE result unless the table is well formed, such as produced in item 3 above. Note that for any well formed table, entry[0] is always 0 and entry[8] is always the polynomial expressed as in item 3 above. The operation can be implemented using shifts of 1, 2, or 4 bits at a time as shown in Table 3-2.

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

7. The CRC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

8. The CRC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

Table 3-1: Common CRC Polynomials

Polynomial	POLY	Initialize Value	Result
CRC-16 (used for DDCMP and Bisync) x^16+x^15+x^2+1	0000A001	00000000	R0<15:0>
CCITT (used for ADCCP, HDLC, SDLC) x^16+x^12+x^5+1	00008408	0000FFFF	one's complement of R0<15:0>
AUTODIN-II x^32+x^26+x^23+x^22+x^16+x^12+ x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1	EDB88320	FFFFFFFF	one's complement of R0<31:0>

Table 3-2: CRC Shift Amounts

Shift Amount(s)	Steps Per Byte (Limit)	Table Index	Table Index Multiplier (i)	Table Entries Used
1	8	tmp3<0>	8	[0]=0,[8]
2	4	tmp3<1:0>	4	[0]=0,[4],[8],[12]
4	2	tmp3<3:0>	1	all

Change History:

Revision J. Rich Brunner, December 1989.

Revision H. Tim Leonard, May 1987.

Revision F. Al Thomas, November 1986.

- Add new instruction-implementation rules.

Revision E. Al Thomas, September 1986.

- Add implied operands for CRC instruction.
- Change the reference citation for CRC algorithms.
- Include CRC as an optional instruction.

Revision D. Tim Leonard, March 1985.

- Change the revision number to correspond to DEC Standard 032 rev number.

Revision 7, repartition Chapter 4. Dileep Bhandarkar, 26 July 1982.

3.12 DECIMAL-STRING INSTRUCTIONS

NOTE

> Decimal-string instructions are optional in implementations of the VAX architecture. Execution of an omitted instruction results in an emulated instruction exception. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11, Implementation Options.

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 two operands:

- The first operand is the length; 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 1).

- The second operand is 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 that 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 5). After the interruption, the instruction resumes transparently. The format of the control block at completion is shown in Figure 3-8. 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.

```
 31                                                            0
+--------------------------------------------------------------+
|                              0                               | :R0
+--------------------------------------------------------------+
|                          address 1                           | :R1
+--------------------------------------------------------------+
|                              0                               | :R2
+--------------------------------------------------------------+
|                          address 2                           | :R3
+--------------------------------------------------------------+
|                              0                               | :R4
+--------------------------------------------------------------+
|                          address 3                           | :R5
+--------------------------------------------------------------+
```

Figure 3-8 Decimal-String-Instruction Control Block

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.

3.12.1 Decimal Overflow

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

3.12.2 Zero Numbers

A zero result has a positive sign for all operations that complete without decimal overflow, except for CVTPT which does not fix a -0 to a +0. When digits are lost because of overflow, however, 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. 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; that is, PSL<N> is cleared and PSL<Z> is set.

3.12.3 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.

3.12.4 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.

3.12.5 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.

3.12.6 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 occurs if an invalid sign is detected. The value of a zero-length leading separate numeric string is identically 0.

ADDP Add Packed

Format:

```
        opcode addlen.rw, addaddr.ab, sumlen.rw,                    4 operand
               sumaddr.ab, {R0-3.wl}

        opcode add1len.rw, add1addr.ab, add2len.rw,
               add2addr.ab, sumlen.rw, sumaddr.ab, {R0-5.wl}       6 operand
```

Operation:

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

        ({sumaddr + ZEXT(sumlen/2)} : sumaddr) <-
                ({add2addr + ZEXT(add2len/2)} : add2addr) +
                ({add1addr + ZEXT(add1len/2)} : add1addr); !6 operand
```

Condition Codes:

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

Exceptions:

reserved operand
decimal overflow

Opcodes:

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

Description:

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

In 6 operand format, the addend 1 string specified by the addend 1 length and addend 1 address operands is added to the addend 2 string specified by the addend 2 length and addend 2 address operands. The sum string specified by the sum length and sum address operands is replaced by the result.

Notes:

1. After execution of ADDP4:

   R0 = 0

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

   R2 = 0

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

2. After execution of ADDP6:

   R0 = 0

   R1 = address of the byte containing the most significant digit of the addend 1 string

   R2 = 0

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

   R4 = 0

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

3. The sum string, R0 through R3 (or R0 through R5 for ADDP6), and the condition codes are UNPREDICTABLE if the sum string overlaps the addend, addend 1, or addend 2 strings; the addend, addend 1, addend 2 or sum (4 operand only) strings contain an invalid nibble; or a reserved operand abort occurs.

4. The ADDPx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

ASHP Arithmetic Shift and Round Packed

Format:

```
        opcode cnt.rb, srclen.rw, srcaddr.ab, round.rb
               dstlen.rw, dstaddr.ab, {R0-3.wl}
```

Operation:

```
        ({dstaddr + ZEXT(dstlen/2)} : dstaddr) <-
               {(({srcaddr + ZEXT(srclen/2)} : srcaddr)
                      + {round <3:0>*{10 ** {-cnt-1}}}}
                      * {10 ** cnt} ;
```

Condition Codes:

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

Exceptions:

reserved operand
decimal overflow

Opcode:

F8	ASHP	Arithmetic Shift and Round Packed

Description:

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

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

Notes:

1. After execution:

   R0 = 0

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

   R2 = 0

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

2. The destination string, R0 through R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string, the source string contains an invalid nibble, or a reserved operand 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 item 4 above.

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

7. The ASHP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

CMPP   Compare Packed

Format:

```
        opcode len.rw, src1addr.ab, src2addr.ab,         3 operand
               {R0-3.wl}

        opcode src1len.rw, src1addr.ab, src2len.rw,
               src2addr.ab, {R0-3.wl}                    4  operand
```

Operation:

```
        ({src1addr + ZEXT(len/2)} : src1addr) -
               ({src2addr + ZEXT(len/2)} : src2addr);      3 operand

        ({src1addr + ZEXT(src1len/2)} : src1addr) -
               ({src2addr + ZEXT(src2len/2)} : src2addr);  4 operand
```

Condition Codes:

```
        N <- {src1 string} LSS {src2 string};
        Z <- {src1 string} EQL {src2 string};
        V <- 0;
        C <- 0;
```

Exception:

        reserved operand

Opcodes:

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

Description:

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

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

Notes:

1. After execution of CMPP3 or CMPP4:

   R0 = 0

R1 = address of the byte containing the most significant digit of string 1

R2 = 0

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

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

3. The CMPPx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

CVTLP Convert Long to Packed

Format:

```
opcode src.rl, dstlen.rw, dstaddr.ab, {R0-3.wl}
```

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

Opcode:

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.

4. The CVTLP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

CVTPL Convert Packed to Long

Format:

```
        opcode srclen.rw, srcaddr.ab, {R0-3.wl}, 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

Opcode:

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 item 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.

6. The CVTPL instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

CVTPS Convert Packed to Leading Separate Numeric

Format:

```
        opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab,
              {R0-3.wl}
```

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

Opcode:

```
  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 PSL<N> bits).

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

6. The CVTPS instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

CVTPT Convert Packed to Trailing Numeric

Format:

```
        opcode srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab,
               {R0-3.wl}
```

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

Opcode:

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

Description:

The source packed decimal string specified by the source length and source address operands is converted to a trailing numeric string. The destination string specified by the destination length and destination address operands is replaced by the result. PSL<N> and PSL<Z> 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 (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, PSL<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 1 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 zeros). 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 PSL<N> bits).

7. The CVTPT instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

CVTSP Convert Leading Separate Numeric to Packed

Format:

```
        opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab,
               {R0-3.wl}
```

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

Opcode:

09	CVTSP	Convert Leading Separate Numeric to Packed

Description:

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

Notes:

1. A reserved operand abort occurs if:

   - The length of the source leading separate numeric string is outside the range 0 through 31.

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

   - 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.

4. The CVTSP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

CVTTP Convert Trailing Numeric to Packed

Format:

```
        opcode srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab,
               {R0-3.wl}
```

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

Opcode:

```
  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 (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:

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

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

- o The source string contains an invalid byte; an invalid byte is any value other than ASCII "0" through "9" in any high-order byte (any byte except the least significant byte)
- o 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, PSL<N> is cleared and PSL<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 1 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.

8. The CVTTP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

DIVP Divide Packed

Format:

```
        opcode divrlen.rw, divraddr.ab, divdlen.rw,
               divdaddr.ab, quolen.rw, quoaddr.ab,
               {R0-5.wl, -16(SP):-1(SP).wb}
```

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
```

Opcode:

```
  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:

   o The absolute value of the remainder (which is lost) is less that the absolute value of the divisor

   o 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

- o 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, dividend, or quotient strings overlap the 16 bytes of temporary storage on the stack, the divisor or dividend string contains an invalid nibble, the divisor is 0, or a reserved operand abort occurs.

5. The DIVP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

MOVP Move Packed

Format:

```
        opcode len.rw, srcaddr.ab, dstaddr.ab, {R0-3.wl}
```

Operation:

```
        ({dstaddr + ZEXT(len/2)} : dstaddr) <-
                ({srcaddr + ZEXT(len/2)} : srcaddr);
```

Condition Codes:

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

Exception:

reserved operand

Opcode:

34	MOVP	Move Packed

Description:

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

Notes:

1. After execution:

   R0 = 0

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

   R2 = 0

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

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

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

4. The MOVP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

MULP Multiply Packed

Format:

```
        opcode mulrlen.rw, mulraddr.ab, muldlen.rw,
               muldaddr.ab, prodlen.rw, prodaddr.ab,
               {R0-5.wl}
```

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

Opcode:

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.

3. The MULP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

SUBP Subtract Packed

Format:

```
        opcode sublen.rw, subaddr.ab, diflen.rw,
               difaddr.ab, {R0-3.wl}                   4 operand

        opcode sublen.rw, subaddr.ab, minlen.rw,
               minaddr.ab, diflen.rw, difaddr.ab,      6 operand
               {R0-5.wl}
```

Operation:

```
        ({difaddr + ZEXT(diflen/2)} : difaddr) <-
                ({difaddr + ZEXT(diflen/2)} : difaddr) -
                ({subaddr + ZEXT(sublen/2)} : subaddr); !4 operand

        ({difaddr + ZEXT(diflen/2)} : difaddr) <-
                ({minaddr + ZEXT(minlen/2)} : minaddr) -
                ({subaddr + ZEXT(sublen/2)} : subaddr); !6 operand
```

Condition Codes:

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

Exceptions:

reserved operand
decimal overflow

Opcodes:

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

Description:

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

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

Notes:

1. After execution of SUBP4:

   R0 = 0

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

   R2 = 0

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

2. After execution of SUBP6:

   R0 = 0

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

   R2 = 0

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

   R4 = 0

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

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

4. The SUBPx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

Change History:

Revision J. Rich Brunner, December 1989.

Revision H. Tim Leonard, May 1987.

Revision F. Al Thomas, November 1986.
- o Add new instruction-implementation rules.

Revision E. Al Thomas, September 1986.
- o Add implied operands to instruction descriptions.
- o Include the decimal string instructions as optional.

Revision D. Tim Leonard, March 1985.
- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Note that decimal instructions may be omitted, and emulation may use the stack.
- o If DIVP operands overlap the 16 bytes of temporary storage on the stack, the results are UNPREDICTABLE.

Revision 7.1, MicroVAX update. Dileep Bhandarkar, 26 June 1983.
Revision 7. Dileep Bhandarkar, 26 July 1982.
Revision 6, typos and clarifications. Dileep Bhandarkar, 12 May 1980.
- o Change reserved operand faults to aborts
- o Clarify decimal overflow.
- o CVTPT does not fix up -0 to +0.

Revision 5. Dileep Bhandarkar, 2 November 1978.
Revision 4, add new string instructions and packed decimal standard. Tom Rarich, 15 February 1977.
- o Change Numeric string to Trailing Numeric.
- o Change CVTNP and CVTPN to CVTTP and CVTPT.
- o Add Leading Separate String converts CVTSP,CVTPS.
- o ASHP now treats -0 overflow like all other instructions.
- o MULP and DIVP now produce +0 unless decimal overflow occurs.
- o All operations now produce +0 unless decimal overflow occurs.
- o Define reserved operand and overflow handling once at beginning of section.
- o Add note on convert to and from longword to discuss use of registers and overlap.
- o Add clarifying notes at the beginning of the section on zero length strings.
- o Add note on overlap with CMPP instruction.
- o Change zoned to packed operations (decimal data ECO).
- o Clarify -0 in all results and sources (decimal data ECO attachment).
- o Clarify overflow as non-zero digits, not significant digits.
- o Leave CVTLP address in R3 not R1.
- o ASHP changed to include rounding operand.

Revision 3, ECOs 12 through 18, results of the April Task Force review, and the May 25 meeting. Bill Strecker, 10 June 1976.
- o Reserved operand aborts become faults.

- o Add empty section for EDITN.
- o Change pointer to longword or address; make it 32 bits.
- o Add MINU function in ISP.
- o Explicitly give SEXT or ZEXT in all cases needed.
- o Specified condition codes on all exceptions.
- o Lower limit on decimal string length is 0.
- o Digits not checked by numeric instructions.
- o Remove MULN4, DIVN4, and change names to MULN, DIVN.
- o Clobber R2, R3 in CVTLN, CVTNL.
- o CVTNL returns correctly signed low order bits of result on overflow.
- o Remove CVTLU, CVTPU, ASHU, and change names of CVTLS, CVTPS, ASHS to CVTLN, CVTPN, ASHN.
- o Split into separate specifications.

Revision 2, ECOs 1 through 11. Bill Strecker, 16 March 1976.
Revision 1, initial distribution. Bill Strecker, 25 September 1975.

3.13 EDIT INSTRUCTION

NOTE

> The edit instruction is optional in implementations of the VAX architecture. Execution of the instruction, when omitted, results in an emulated instruction exception. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11, Implementation Options.

The edit instruction is designed to implement the common editing functions for handling fixed-format output. The instruction converts an input packed decimal number to an output character string, generating characters for the output. 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. 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 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 that is interpreted in much the same way as are the normal instructions. 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 register contains either an ASCII blank or a minus sign depending upon the sign of the input. The value of the register 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). PSL<N> is determined at the start of the instruction and is not changed thereafter (except to correct a -0 input). The other condition codes are computed and updated as the instruction proceeds. When the EDITPC instruction terminates, registers R0 through 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,
        {R0-5.wl}
```

Operation:

```
        if srclen GTRU 31 then {reserved operand abort};
        PSW<V,C> <- 0;
        PSW<Z> <- 1;
        PSW<N> <- {src has minus sign};
        R0 <- srclen;
        tmp1 <- R0;
        R1 <- srcaddr;
        R2<15:8> <- {if PSW<N> EQL 0 then " " else "-"} ! sign of src
                        !R2<7:0> is used for the fill character
        R3 <- pattern;
        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 fault}};
                {perform pattern operator};
                if NOT exit_flag then {increment R3};
                end;

        if R0 NEQ 0 then {reserved operand abort};
        R0 <- tmp1;             !length of source string
        R1 <- R1 - {tmp1/2}     !point to start of source string
        R2 <- 0;
        R4 <- 0;
        if PSW<Z> EQL 1 then PSW<N> <- 0;
```

Condition Codes:

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

Exceptions:

```
        reserved operand
        decimal overflow
```

Opcode:

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 (EO$END) pattern operator is encountered. The pattern string consists of one-byte pattern operators. Some pattern operators take no operands. Some take a repeat count which is contained in the right-most nibble of the pattern operator itself. The rest take a one-byte operand which immediately follows the pattern operator. This operand is either an unsigned integer length or a byte character. The individual pattern operators are described in Tables 3-3 and 3-4, and on the following pages.

Notes:

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

2. The destination string is UNPREDICTABLE if the source string contains an invalid nibble, if the EO$ADJUST_INPUT operand is outside the range 1 through 31, if the source and destination strings overlap, or if the pattern and destination strings overlap.

3. After execution, the registers are as shown in Figure 3-9. If the destination string is UNPREDICTABLE, R0 through R5 and the condition codes are UNPREDICTABLE.

4. If PSL<V> is set at the end and PSL<DV> is enabled, numeric overflow trap occurs unless the conditions in item 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 up to one page of 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 UNPREDICTABLE.

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 UNPREDICTABLE.

10. On an unimplemented or reserved pattern operator, a reserved operand fault is taken with R3 = location of the faulting pattern operator. This fault may be continued as long as the defined register state is manipulated according to the pattern operator description and the state specified as implementation dependent is preserved. FPD is set and the condition codes and registers are as follows:

    N = {src has minus sign}

    Z = all source digits 0 so far

    V = non-zero digits lost

    C = significance

    R0<31:16> = -{count of source zeros to supply}
    R0<15:0> = remaining srclen<15:0>

    R1 = current source location

    R2<31:16> = implementation dependent
    R2<15:8> = current contents of sign character register
    R2<7:0> = current contents of fill character register

    R3 = location of edit pattern operator causing exception

    R4 = implementation dependent

    R5 = location of next destination byte

11. The EDITPC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11, Implementation Options.

12. The EDITPC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

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

A, X	Use MOVC and MOVB.
PL/I: E, K	(Floating point.) Separate into two integers.
PL/I: I, R, T	(Overpunch.) Treat as 9 and fix up afterwards.
V, P, PL/I: F	Scale by ASHP to get correct position first.
BASIC: %	Special case code.
BASIC: C, L, R, E	MOVC with special code.
FORTRAN: *	Special code triggered by overflow.
FORTRAN: leading -	Extra byte in destination string.

\

```
 31                            16 15                           0
+-------------------------------+-------------------------------+
|                               |         source length         | :R0
+-------------------------------+-------------------------------+
|                        source address                         | :R1
+-------------------------------+-------------------------------+
|                               0                               | :R2
+-------------------------------+-------------------------------+
|            address of EO$END pattern operator                 | :R3
+-------------------------------+-------------------------------+
|                               0                               | :R4
+-------------------------------+-------------------------------+
| address of one byte past the last byte of destination string  | :R5
+---------------------------------------------------------------+
```

Figure 3-9 EDITPC Control Block

Table 3-3: EDIT Pattern Operators

Class	Name	Operand	Summary
insert	EO$INSERT	char	insert char, fill if insignificant
	EO$STORE_SIGN	none	insert sign
	EO$FILL	r	insert fill
move	EO$MOVE	r	move digits, filling insignificant
	EO$FLOAT	r	move digits, floating sign
	EO$END_FLOAT	none	end floating sign
fixup	EO$BLANK_ZERO	len	fill backward when zero
	EO$REPLACE_SIGN	len	replace with fill if -0
load	EO$LOAD_FILL	char	load fill character
	EO$LOAD_SIGN	char	load sign character
	EO$LOAD_PLUS	char	load sign character if positive
	EO$LOAD_MINUS	char	load sign character if negative
control	EO$SET_SIGNIF	none	set significance flag
	EO$CLEAR_SIGNIF	none	clear significance flag
	EO$ADJUST_INPUT	len	adjust source length
	EO$END	none	end edit

Key: char = one character
      r   = repeat count in the range 1 through 15
      len = length in the range 1 through 255

Table 3-4: EDIT Pattern-Operator Encoding

Encoding	Operator	Operand
00	EO$END	none
01	EO$END_FLOAT	none
02	EO$CLEAR_SIGNIF	none
03	EO$SET_SIGNIF	none
04	EO$STORE_SIGN	none
05..1F	Reserved to DIGITAL	
20..3F	Reserved for all time	
40	EO$LOAD_FILL	char is in next byte
41	EO$LOAD_SIGN	char is in next byte
42	EO$LOAD_PLUS	char is in next byte
43	EO$LOAD_MINUS	char is in next byte
44	EO$INSERT	char is in next byte
45	EO$BLANK_ZERO	unsigned length is in next byte
46	EO$REPLACE_SIGN	unsigned length is in next byte
47	EO$ADJUST_INPUT	unsigned length is in next byte
48..5F	Reserved to DIGITAL	
60..7F	Reserved to DIGITAL's customers	
80	Reserved to DIGITAL	
81..8F	EO$FILL	repeat count is <3:0>
90	Reserved to DIGITAL	
91..9F	EO$MOVE	repeat count is <3:0>
A0	Reserved to DIGITAL	
A1..AF	EO$FLOAT	repeat count is <3:0>
B0..FE	Reserved to DIGITAL	
FF	Reserved for all time	

On the following pages, each pattern operator is defined in a format similar to that of 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 EO$ADJUST_INPUT
                end;
        else
                begin
                READ <- (R1)<3+4*R0<0>:4*R0<0>>; !get next nibble
                                        !alternating high then low
                R0 <- R0 - 1;
                if R0<0> EQL 1 then R1 <- R1 + 1;
                end;
        return;


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

Also the following definitions are used:

```
        fill =  R2<7:0>

        sign =  R2<15:8>
```

EO$ADJUST_INPUT Adjust Input Length

Purpose:

Handle source strings with lengths different from the output

Format:

```
        pattern    len
```

Operation:

```
        if len EQLU 0 or len GTRU 31 then {UNPREDICTABLE};
        if R0<15:0> GTRU len
        then
                begin
                R0<31:16> <- 0
                repeat R0<15:0> - len do
                        if READ NEQU 0 then
                                begin
                                PSW<Z> <- 0;
                                PSW<C> <- 1;    !set significance
                                PSW<V> <- 1;
                                end;
                end;
        else R0<31:16> <- R0<15:0> - len; !negative of number to fill
```

Pattern operators:

47 EO$ADJUST_INPUT Adjust Input Length

Description:

The pattern operator is followed by an unsigned byte integer length in the range 1 through 31. If the source string has more digits than this length, the excess 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$BLANK_ZERO Blank Backwards When Zero

Purpose:

Fix the destination to be blank when the value is zero

Format:

```
        pattern     len
```

Operation:

```
        if len EQLU 0 then {UNPREDICTABLE};
        if PSW<Z> EQL 1 then
                begin
                R5 <- R5 - len;
                repeat len do STORE(fill);
                end;
```

Pattern operators:

45	EO$BLANK_ZERO	Blank Backwards When Zero

Description:

The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero, then the contents of the fill register are 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 up to one page of 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$END End Edit

Purpose:

End the edit operation

Format:

```
        pattern
```

Operation:

```
        exit_flag <- true;      !terminate edit loop
                                !end processing is
                                !described under EDITPC instruction
```

Pattern operators:

```
  00    EO$END          End Edit
```

Description:

The edit operation is terminated.

Notes:

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

2. If the source value is -0, PSL<N> is cleared.

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 (that is, if significance is not set), the contents of the sign register are 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$INSERT and EO$FILL pattern operators.

EO$FILL Store Fill

Purpose:

Insert the fill character

Format:

```
        pattern     r
```

Operation:

```
        repeat r do STORE(fill);
```

Pattern operators:

8x	EO$FILL	Store Fill

Description:

The right nibble of the pattern operator is the repeat count. The contents of the fill register is placed into the destination repeat times.

Notes:

This pattern operator is used for fill (blank) insertion.

EO$FLOAT Float Sign

Purpose:

Move digits, floating the sign across insignificant digits

Format:

```
        pattern     r
```

Operation:

```
        repeat r do
                begin
                tmp <- READ;
                if tmp NEQU 0 then
                        begin
                        if PSW<C> EQL 0 then
                                begin
                                STORE(sign);
                                PSW<Z> <- 0;
                                PSW<C> <- 1;    !set significance
                                end;
                        end;
                if PSW<C> EQL 0 then STORE(fill)
                        else STORE("0" + tmp);
                end;
```

Pattern operators:

Ax	EO$FLOAT	Float Sign

Description:

The right nibble of the pattern operator is the repeat count. For repeat times, the following algorithm is executed. The next digit from the source is examined. If it is non-zero and significance is not yet set, then the contents of the sign register are stored in the destination, significance is set, and zero is cleared. If the digit is significant, it is stored in the destination; otherwise, the contents of the fill register 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$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 are placed into the destination.

Notes:

This pattern operator is used for blankable inserts (comma, for example) and fixed inserts (slash, for example). Fixed inserts require that significance be set (by EO$SET_SIGNIF or EO$END_FLOAT).

EO$LOAD_   Load Register

Purpose:

Change the contents of the fill or sign register

Format:

```
        pattern     ch
```

Operation:

```
                        !select one depending on pattern operator
        fill <- ch;             !EO$LOAD_FILL

        sign <- ch;             !EO$LOAD_SIGN

        if PSW<N> EQL 0 then sign <- ch;        !EO$LOAD_PLUS

        if PSW<N> EQL 1 then sign <- ch;        !EO$LOAD_MINUS
```

Pattern operators:

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

Description:

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

Notes:

1. EO$LOAD_FILL is used to setup check protection (asterisk 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$MOVE Move Digits

Purpose:

Move digits, filling for insignificant digits (leading zeros)

Format:

```
        pattern     r
```

Operation:

```
        repeat r do
                begin
                tmp <- READ;
                if tmp NEQU 0 then
                        begin
                        PSW<Z> <- 0;
                        PSW<C> <- 1;    !set significance
                        end;
                if PSW<C> EQL 0 then STORE(fill)
                        else STORE("0" + tmp);
                end;
```

Pattern operators:

9x	EO$MOVE	Move Digits

Description:

The right nibble of the pattern operator is the repeat count. For repeat times, the following algorithm is executed. The next digit is moved from the source to the destination. If the digit is non-zero, significance is set and zero is cleared. If the digit is not significant (that is, if it is a leading zero), it is replaced by the contents of the fill register in the destination.

Notes:

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

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

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

EO$REPLACE_SIGN Replace Sign When Zero

Purpose:

Fix the destination sign when the value is zero

Format:

```
        pattern     len
```

Operation:

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

Pattern operators:

46	EO$REPLACE_SIGN Replace Sign When Zero

Description:

The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero (that is, if PSL<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 up to one page of memory preceding it are UNPREDICTABLE.

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

EO$_SIGNIF Significance

Purpose:

Control the significance (leading zero) indicator

Format:

```
        pattern
```

Operation:

```
        PSW<C> <- 0;            !EO$CLEAR_SIGNIF

        PSW<C> <- 1;            !EO$SET_SIGNIF
```

Pattern operators:

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

Description:

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

Notes:

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

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

EO$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 are placed into the destination.

Notes:

This pattern operator is used for any non-floating arithmetic sign. It should be preceded by a EO$LOAD_PLUS or EO$LOAD_MINUS if the default sign convention is not desired.

Change History:

Revision J. Rich Brunner, December 1989.

Revision H. Tim Leonard, May 1987.

Revision F. Al Thomas, November 1986.
Add new instruction-implementation rules.

Revision E. Al Thomas, September 1986.
- o Add implied operands for EDITPC instruction.
- o Include EDITPC as an optional instruction.

Revision D. Tim Leonard, March 1985.
- o Note that EDITPC may be omitted, and emulation may use stack space.
- o Change the revision number to correspond to DEC Standard 032 rev number.
- o If pattern operator specifies length out of range, up to one page of memory is UNPREDICTABLE.

Revision 7, typos. 26 July 1982.
- o Srclen > 31 is a reserved operand abort, not fault.

Revision 6, ECOs. 12 May 1980.
- o ECO to EO$FLOAT.
- o ECO to EO$REPLACE_SIGN.

Revision 5, reissue. 2 November 1978.
Revision 4, prune. Peter Conklin, 11 April 1977.
- o Drop insert zero/slash and skip zero/slash because infrequent.
- o Combine fixed inserts and insert or protect. Add one byte EO$SET_SIGNIF.
- o Drop move numeric or fill (can use EO$MOVE with significance clear).
- o Combine move into EO$MOVE by setting significance.
- o Drop insert protection register.
- o Drop insert comma/period as optimizations not worth it now.
- o Drop byte repeats. Make repeats be 0 reserved and 1..15 in low nibble.
- o Require EO$END instead of running off the end.
- o Combine protection and fill into one register.
- o Combine currency and sign into one register. This combines the logic of floating currency and sign.
- o Drop optimized combination of EO$BLANK_ZERO and EO$END.
- o Replace options and sign testing fills with EO$LOAD_PLUS and EO$LOAD_MINUS.

  The above pruning results in an average growth in the length of an edit pattern string of about 4 bytes. This is offset by removing two operands (typically 2 to 4 bytes).
- o Add EO$REPLACE_SIGN to fixup -0. This saves about 10 bytes and several operators over the previous proposal.

- o Confine the pattern to associate with the field. Drop destination length operand and adjust destination.
- o Drop input editing--this would be a distinct instruction when needed in volume. Change name to EDITPC.
- o Map flags onto the condition codes.
- o Do not touch R6 and R7.
- o Drop flag set/clear/branch and position add/subtract.
- o Invert sense of significance flag.
- o Assign pattern op codes.
- o Make 0 length of EO$BLANK_ZERO and EO$REPLACE_SIGN be UNPREDICTABLE.
- o Use the term "abort" rather than "fault" when wrong number of input digits.
- o Change "UNDEFINED" to "UNPREDICTABLE".
- o Add usage under notes.
- o Restrict EO$ADJUST_INPUT to 31 digits.
- o Set R0-R1 at end to point to input string.
- o Correct ISP to not increment pattern pointer at end.
- o Add comments to pseudo ISP to aid the initial reader.

Revision 3, complete the design. Peter Conklin, May 1976.

- o Add option to set protection register to blank vs. asterisk. reason: removes unnecessary duplication of operators.
- o Add operator EOMNF (move numeric or fill). reason: PL/I Y-picture.
- o Add option to force the sign during initialization. reason: handle special sign conventions.
- o Add operator EOMNFS (move numeric or floating sign). reason: handle FORTRAN sign convention.
- o Use R6, R7 to keep extra EDITN context. Reason: keeps left half of counts for normal string contents.
- o Change name to EDITN.
- o Add ability to handle variable length input and output strings.
- o Change to zoned sign and replace EFSS with EFSO.
- o Keep lengths GTRU 0.
- o Add skip numeric to validate numeric string and skip alphabetic for consistency.
- o Support packed format.
- o Limit edit character string to 255 bytes. This drops all word sizes.
- o Drop move numeric or minus--can be done by selecting one of two patterns.

Revision 2, reduce context. Peter Conklin, February 1976.

- o Combine two instructions into one. Make an initialization option whether or not to examine the input for the presence of a sign. reason: drop operation code.
- o Replace BWZF flag with a pattern operator to protect backward when zero. reason: decrease context; fewer memory cycles.
- o Reduce context to 4 bytes of edit registers, 5 flags, and only one hidden count. reason: reduce context at no loss of function.
- o Replace pattern length operand with initialization operand. reason: reduce context.

- o Add specification of handling when input and output lengths do not match.
- o Avoid nibble-sized counts. Replace with byte-sized. reason: consistency with regular instruction set.
- o Reference edit registers by special operators rather than generic mechanics. reason: support of point 3.
- o Replace flag nibble with general set and clear of any flag. reason: generalize and simplify.
- o Add pattern operators to adjust source and destination pointers. reason: generalize and simplify.
- o Set condition codes to reflect the result. reason: consistency with the rest of the architecture.
- o Add input conversion operators. reason: completeness. They are optionally subsettable.
- o On numeric moves and edits, if the input is not a blank or digit, set conversion error. If non-zero, clear Zero flag. reason: correct setting of condition codes.
- o If input minus sign and not output, then set conversion error. reason: correct setting of condition codes.
- o Add specification of behavior on reserved pattern operator. reason: allow simulation of missing operators.
- o Use fill or protection registers for all blanking. reason: flexibility.

Revision 1, initial proposal. Blair, August 1975.

CHAPTER 4

MEMORY MANAGEMENT

Memory management consists of the hardware and software that control the allocation and use of physical memory. The effect of memory management is exemplified in a multiprogramming system where several processes may reside in physical memory at the same time. To ensure that one process will not affect other processes or the operating system, VAX architecture uses memory protection and multiple address spaces.

Four hierarchical access modes provide the memory access control, which further improves software reliability. These access modes are, from most to least privileged, kernel, executive, supervisor, and user. For each of the four access modes, protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write. 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.

Memory management provides the CPU with mapping information. First, the CPU generates virtual addresses when an image is executed. Before these addresses can be used to access instructions and data, however, 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 uses this mapping information when it translates virtual addresses to physical addresses.

Memory management, then, is the scheme that provides both the memory protection and memory-mapping mechanisms of VAX architecture. Memory management accomplishes the following:

- Provides a large address space for instructions and data
- Allows data structures up to one gigabyte
- Provides convenient and efficient sharing of instructions and data
- Contributes 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 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 with 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.

4.1 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, sharing, and protection.

This virtual address space is too large to be contained in generally 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 saves the inactive but used parts of the virtual address space on the external storage media.

The virtual address space is divided into two parts, per-process space and system space, discussed in the following sections. Virtual address space is illustrated in Figure 4-1.

4.1.1 Process Space

The half of the virtual address space with smaller addresses (addresses 00000000 through 7FFFFFFF, hex) is termed per-process space. 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 potentially completely disjoint. The address map for per-process space is context switched (changed) when the process running on the system is changed (see Chapter 6, Process Structure).

```
                  +--------------------------------+
0000 0000:        |                                |  \
                  |      P0 (program) region       |   |
                  |                                |   |
                  |- - - - - -P0 length- - - - - -|   |
                  | |                              |   |
                  | V  P0 region growth direction |   |
3FFF FFFF:        |                                |   |  per-process
                  +--------------------------------+   |  space
4000 0000:        |                                |   |
                  | ^  P1 region growth direction |   |
                  | |                              |   |
                  |- - - - - -P1 length- - - - - -|   |
                  |                                |   |
                  |      P1 (control) region       |   |
7FFF FFFF:        |                                |  /
                  +--------------------------------+
8000 0000:        |                                |  \
                  |         system region          |   |
                  |                                |   |
                  |- - - - -system length- - - - -|   |
                  | |                              |   |
                  | V system region growth        |   |
BFFF FFFF:        |   direction                    |   |  system
                  +--------------------------------+   |  space
C000 0000:        |                                |   |
                  |                                |   |
                  |                                |   |
                  |        reserved region         |   |
                  |                                |   |
                  |                                |   |
FFFF FFFF:        |                                |  /
                  +--------------------------------+
```

Figure 4-1 Virtual-Address Space

```
 31 30 29                                     9 8              0
+-----+--------------------------------------+-----------------+
| reg |          virtual page number         | byte within page|
+-----+--------------------------------------+-----------------+
```

Figure 4-2 Virtual-Address Format

4.1.2 System Space

The half of virtual address space with larger addresses (addresses 80000000 through FFFFFFFF, hex) 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.

4.1.3 Virtual Address Format

The VAX 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. When memory management is enabled, the virtual address consists of a region field, a virtual page number (VPN) field, and a byte within page field, as shown in Figure 4-2 and described below.

The VPN field, bits <29:9> of a virtual address, specifies the virtual page to be referenced. The virtual address space contains 8,388,608 (2**23) pages. The byte-within-page field, bits <9:0> of a virtual address, specifies the byte offset within the page. A page contains 512 bytes.

The region field (bits <31:30> of a virtual address) is part of the virtual page number and specifies which of four regions the virtual address references. When bit <31> of a virtual address is 1, the address is in the system space. When bit <31> is 0, the address is in the per-process space.

Within system space, bit <30> distinguishes between the system region and a reserved region. When bits <31:30> are 11 (binary), the address refers to the reserved region. When bits <31:30> are 10 (binary), the address refers to the system region.

Within per-process space, bit <30> distinguishes between the program and control regions. When bits <31:30> are 01 (binary), the control region is referenced; and when bits <31:30> are 00, the program region is referenced.

4.1.4 Virtual Address Space Layout

The layout of virtual address space is illustrated in Figure 4-1. Note that access to each of the three regions (P0, P1, system) is controlled by a length register (P0LR, P1LR, SLR). Within the limits set by the length registers, the access is further controlled by page tables that specify the validity, access requirements, and physical location of each page in the memory.

4.1.5 Address Space Numbers (ASN)

On some processors, an individual process address space can be identified by an address space number (ASN), an arbitrary three-byte identifier which the operating system specifies. Such processors maintain the ASN of the process address space provided to the process in the ASN register. This register is four bytes wide with the three high-order bytes containing the identifier and the low-order byte ignored. The ASN register is part of a process's hardware context and is written by both MTPR and LDPCTX and read by MFPR. It is illustrated in figure 4-3. The value of the ASN register is associated with per-process TB state; see section 4.7, Translation Buffer, for more details. At processor initialization time, the contents of this register are UNPREDICTABLE. If a processor implements ASN, it also implements CPUID, TBIASN, and PCB<PRVCPU>.

An operating system should assign a unique ASN to each process address space. As a consequence, if an operating system provides a unique process address space to each process, then no two processes will have identical values in their ASN registers. When an operating system places the same value in the ASN registers of multiple processes, it is indicating to the processor that these processes have been provided with the same process address space and so can share process TB state \which includes process TB and virtual cache entries\.

\Note: "Process address space" is a confusing term. It does not indicate that an address space belongs to a process; rather it indicates that the virtual addresses within the address space range from 0 to 7ffffff. Terminology such as this will be removed or clarified in another ECO to come.\

```
 3
 1                                         8 7               0
+-----------------------------------------+-----------------+
|       Address Space Number              |ignored;returns 0|
+-----------------------------------------+-----------------+
```

Figure 4-3 Address Space Number Register (ASN)

4.2 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 4-4 illustrates the privileged map-enable register.

MAPEN<0> is the memory-mapping-enable 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. Software should follow a MTPR-to-MAPEN with an REI instruction. (For further details see section 7.2.2.)

NOTE

\The MTPR #1,#PR$_MAPEN to enable memory management need not flush the instruction buffer. In order to implement the console CONTINUE command, the MicroVAX console uses the following code:

```
.align long
MTPR   #1,#PR$_MAPEN
REI
```

The .align directive forces the MTPR and REI into the same longword which means that both instructions are fetched while memory management is still disabled. If the MTPR were to flush the instruction buffer, the fetch for the REI would be affected by the new MAPEN state.\

4.2.1 Memory Management Disabled

When software disables memory mapping (by setting MME to 0), the processor turns off access control and translates virtual addresses to physical addresses either by truncation, sign extension, or one-to-one mapping depending on the physical address mode implemented by the processor.

If the physical address mode that is implemented is 24-bit or 30-bit mode, virtual addresses are converted to 24-bit and 30-bit physical addresses respectively by truncating the high-order bits.

If the physical address mode is 32-bit one-to-one mode, the entire 32-bit virtual address is used as a 32-bit physical address. If the physical address mode is 32 bit sign-extended mode, a virtual address is converted by an implementation-dependent function, which is preferably to sign extend VA bits <29:0> to 32 bits, and only part of the physical address space is visible when mapping is disabled. Whether a processor uses 32-bit one-to-one mode or 32-bit sign-extended mode depends on an implementation-dependent function, such as a mode-control bit.

If the physical address mode is 34 bit, virtual addresses are converted by an implementation-dependent function, and only part of the physical address space is visible when mapping is disabled. The preferred implementation of the 32-to-34-bit mapping is sign extension of VA bits <29:0>.

The specific implementation of a number of memory management features is determined by the physical address mode that the processor supports. These features are described in subsequent sections and chapters. For convenience, the table at the end of this section shows which implementation features are allowed for each physical address mode.

When memory management is disabled, there is no page protection: all accesses are allowed in all modes. In addition, no modify bit is maintained, and no memory management faults occur.

The results of a reference to nonexistent memory are implementation specific. \The preferred implementations are machine check or an error interrupt.\ References to addresses beyond the end of the physical address space result in UNDEFINED behavior. For example, a VAX-11/730, which implements a 24-bit physical address, may have 1 megabyte of memory installed. On such a system, a reference to an address in the first megabyte will read or write memory; a reference to an address in the second megabyte will result in a "nonexistent memory" machine-check exception; and a reference to an address in the 17th megabyte will result in UNDEFINED processor operation.

Probing nonexistent memory (or addresses beyond the end of the physical address space) results in UNDEFINED processor operation. In addition, a processor may have an instruction buffer that prefetches instructions before execution. If the instruction stream comes within 512 bytes of nonexistent memory or the end of the physical address space when memory management is disabled, prefetcher references may cause UNDEFINED behavior. See Chapter 7 for additional information about physical memory and its operation.

Physical Address Mode:

	24-bit	30-bit	32-bit sign-extended	32-bit one-to-one	34-bit
Address Size	PA<23:0>	PA<29:0>	PA<31:0>	PA<31:0>	PA<33:0>
Generated from	VA<23:0>	VA<29:0>	SEXT(VA<29:0>)	VA<31:0>	SEXT(VA<29:0>)
PFN size:	21-bit	21-bit	21-bit	25-bit	25-bit
Address in SBR	Phys	Phys	Phys	Phys	Phys
Address in SCBB	Phys	Phys	Phys	Phys	Phys/Virt
Address in PCBB	Phys	Phys/Virt	Phys	Phys	Virt
Range of SBR, SCBB, and PCBB used by processor	<23:0>	<29:0>	<29:0>	<31:0>	<29:0>

4.3 ADDRESS TRANSLATION

When software enables memory mapping (by setting MME to 1), the processor turns access control on and translates virtual addresses to physical addresses by using page tables in memory. The processor uses the following to determine whether an intended access is allowed:

- The virtual address, which is used to index a page table
- The intended access type (read or write)
- 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 page table entry is valid), 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.

4.3.1 Page Table Entry

The processor uses a page table entry (PTE) to translate virtual addresses to physical addresses. For a VAX processor with a physical address mode of either 24-bit, 30-bit, or 32-bit sign-extended, the PTE format includes a 21-bit page-frame-number (PFN) field, as illustrated in Figure 4-6 and described in Table 4-2. For a VAX processor with a physical address mode of 32-bit one-to-one or 34-bit, the PTE format includes a 25-bit PFN field, as illustrated in Figure 4-5 and described in Table 4-1.

When PTE<V> (the PTE valid bit, bit<31>) is zero, the PTE is said to be invalid. If software makes a reference to a page mapped by an invalid PTE (and the PTE protection field allows the access), the processor initiates a translation-not-valid fault. 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 paged-out states. VAX/VMS encodes these functions in invalid PTEs and processes them whenever a page fault occurs.

The processor maintains (or helps software maintain) a record of pages that have been modified. When software attempts to write a page that is mapped by a valid PTE that allows the access, but PTE<M> (the modify bit, bit<26>) is zero, the processor either initiates a modify fault and flushes the faulting PTE from the TB, or sets PTE<M> and continues, depending on the implementation. For process-space pages, only a zero value for process PTE<M> causes the modify fault or the setting of process PTE<M>; the value of system PTE<M> is ignored. Processors neither set PTE<M> nor initiate modify fault if the page's protection denies access or if the PTE is invalid.

Processors that implement the virtual-machine option also implement the modify-fault option.

Processors that implement modify fault never write the PTE. (The exception is the VAX 8800 family when the virtual-machine option is implemented: PTE<M> is set for the reported page, and only the reported page, if the page is valid and writable.)

Processors that do not implement modify fault set PTE<M> as part of the result of a successful write or modify to a page (when mapping is enabled). Such processors may also set PTE<M> as part of the result of a probe-write instruction (PROBEW) or as part of an implied probe-write.

When an instruction fault occurs on a processor that sets PTE<M> (other than a modify fault on a VAX 8800 family processor), it is UNPREDICTABLE whether or not the PTE<M> bits are set for the valid and writable pages containing the instruction's write operands. 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. The processor does not set M in the PTE mapping the first page. It is UNPREDICTABLE whether the processor sets M in the PTE mapping the second page.

It is UNPREDICTABLE whether the processor's setting a process PTE<M> causes the processor to also set M in the system PTE that maps that process PTE.

Note that the update of PTE<M> is not interlocked in a multiprocessor system.

```
 31                                                             1 0
+--------------------------------------------------------------+-+
|                             MBZ                              | |
+--------------------------------------------------------------+-+
```

Figure 4-4 Map Enable Register (MAPEN)

```
 3 3     2 2 2 2
 1 0     7 6 5 4                                                 0
+-+-------+-+-+----------------------------------------------------+
|V| PROT  |M|S|                        PFN                         |
+-+-------+-+-+----------------------------------------------------+
```

Figure 4-5 Page Table Entry (25-Bit Page Frame Number)

Table 4-1: Fields of the PTE (25-Bit Page Frame Number)

Extent	Name	Mnemonic	Meaning
<31>	Valid	V	Indicates the validity of the M bit and PFN field. When V=1, the M and PFN fields are valid for use by hardware; when V=0, they are reserved for DIGITAL software.
<30:27>	Protection	PROT	Indicates at what access modes a process can reference the page. This field is always valid and is used by the CPU hardware even when V=0.
<26>	Modify	M	When V=0, M is not used by CPU hardware and is reserved for DIGITAL software. When V=1, M is used to detect write and modify references to the page.
<25>	Software	S	Reserved for DIGITAL software.
<24:0>	Page Frame Number	PFN	The upper 25 bits of the physical address of the base of the page. Used by CPU hardware only if V=1.

```
 31 30     27 26 25 24 23 22 21 20                                      0
+--+---------+--+--+-----+--+--+-----------------------------------------+
| V|  PROT   | M| Z| OWN | S| S|                 PFN                     |
+--+---------+--+--+-----+--+--+-----------------------------------------+
```

Figure 4-6 Page Table Entry (21-Bit Page Frame Number)

Table 4-2: Fields of the PTE (21-Bit Page Frame Number)

Extent	Name	Mnemonic	Meaning
<31>	Valid	V	Indicates the validity of the M bit and PFN field. When V=1, the M and PFN fields are valid for use by hardware; when V=0, they are reserved for DIGITAL software.
<30:27>	Protection	PROT	Indicates at what access modes a process can reference the page. This field is always valid and is used by the CPU hardware even when V=0.
<26>	Modify	M	When V=0, M is not used by CPU hardware and is reserved for DIGITAL software and I/O devices. When V=1, M is used to detect write and modify references to the page.
<25>	Reserved	Z	Reserved to DIGITAL and must be zero.
<24:23>	Owner	OWN	Reserved for DIGITAL software.
<22:21>	Software	S	Reserved for DIGITAL software.
<20:0>	Page Frame Number	PFN	The upper 21 bits of the physical address of the base of the page. Used by CPU hardware only if V=1.

4.3.2 Changes to Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, VMS sets and clears PTE<V> and changes PTE<PFN> as pages are moved to and from external storage devices.

The software must guarantee that each PTE is always consistent within itself. Changing a PTE one field at a time may give incorrect system operation. An example would be to set PTE<V> with one instruction before establishing PTE<PFN> with another. An interrupt routine between the two instructions could use an address that would map using the inconsistent PTE. The software can solve this problem by building a new PTE in a register and then moving the new PTE to the page table with a single instruction such as MOVL.

Multiprocessing makes the problem more complicated. Another processor, be it another CPU or an I/O processor, can reference the same page tables that the first CPU is changing. The second processor must always read consistent PTEs. In order to guarantee this, two requirements must be met (note that PTEs are longwords, longword-aligned):

1. Whenever the software modifies a PTE in more than one byte, it must use a longword, longword-aligned, and write-destination instruction such as MOVL.

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.

A processor writes a PTE only when setting that PTE's modify bit, and writes the PTE with a longword-aligned longword write.

4.4 MEMORY PROTECTION

Memory protection 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.

4.4.1 Processor Access Modes

In the order of most privileged to least privileged, the four processor modes are:

0	Kernel mode	Page management, scheduling, I/O drivers
1	Executive mode	In VMS, many system-service calls, including RMS
2	Supervisor mode	In VMS, command interpretation
3	User mode	User-level code, utilities, compilers, debuggers

The access mode of a running process is the current processor mode, stored in the current-mode field of the processor status longword (PSL).

4.4.2 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 the sense that all processes see the same system space, a program may be prevented from modifying or even reading portions of it. A program may also be prevented from reading or modifying portions of per-process space.

In system space, for example, 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 space 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:

- Each mode's access can be read-write, read-only, or no-access.

- If any level has read access, then all more privileged levels also have read access.

- If any level has write access, then all more privileged levels also have write access.

- If any level has write access, then it also has read access.

The protection codes for the 15 combinations of page protection are encoded in a 4-bit field in the page table entry, as shown in Table 4-3.

\The encoding scheme shown in Table 4-3 was chosen to simplify hardware access checking for implementations not using a table decoder. The access is allowed if:

```
    {CODE NEQU 0} AND
        {{CODE EQLU 4} OR {CM LSSU WM} OR {READ AND {CM LEQU RM}}}

        CM is current processor mode
        RM is left 2 bits of code
        WM is one's complement of right 2 bits of code\
```

Table 4-3: PTE Protection Codes

				Accessibility			
Name	Mnemonic	Decimal	Binary	Kernel	Exec	Super	User
no access	NA	0	0000	none	none	none	none
reserved		1	0001	UNPREDICTABLE			
kernel write	KW	2	0010	write	none	none	none
kernel read	KR	3	0011	read	none	none	none
user write	UW	4	0100	write	write	write	write
exec write	EW	5	0101	write	write	none	none
exec read / kernel write	ERKW	6	0110	write	read	none	none
exec read	ER	7	0111	read	read	none	none
super write	SW	8	1000	write	write	write	none
super read / exec write	SREW	9	1001	write	write	read	none
super read / kernel write	SRKW	10	1010	write	read	read	none
super read	SR	11	1011	read	read	read	none
user read / super write	URSW	12	1100	write	write	write	read
user read / exec write	UREW	13	1101	write	write	read	read
user read / kernel write	URKW	14	1110	write	read	read	read
user read	UR	15	1111	read	read	read	read

write - read and write
read - read only
none - no access

```
 31 30 29                                     9 8              0
+--+--+---------------------------------------+----------------+
| 1| 0|      virtual page number              | byte within page|
+--+--+---------------------------------------+----------------+
```

Figure 4-7 System Virtual-Address Format

4.4.3 Length Violation

Every valid virtual address lies within bounds determined by the addressing region (P0, P1, or system) and the contents of the length register associated with that region (P0LR, P1LR, or SLR). A reference to an address outside these bounds causes a length-violation fault. The addressing bounds algorithm is a simple limit check whose formal notation is:

```
case VAddr<31:30>
    set
    [0]:                                    ! P0 region
        if ZEXT( VAddr<29:9> ) GEQU P0LR
            then {length violation};
    [1]:                                    ! P1 region
        if ZEXT( VAddr<29:9> ) LSSU P1LR
            then {length violation};
    [2]:                                    ! System region
        if ZEXT( VAddr<29:9> ) GEQU SLR
            then {length violation};
    [3]:                                    ! reserved region
        {length violation};
    tes;
```

4.4.4 Access-Control-Violation Fault

An access-control-violation 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.

4.4.5 Access Across a Page Boundary

If an access is made across a page boundary, the order in which the pages are accessed is UNPREDICTABLE. For a single reference to a page, however, access-control-violation fault always takes precedence over translation-not-valid fault and modify fault.

4.5 SYSTEM-SPACE ADDRESS TRANSLATION

A virtual address with <31:30> = 2 is an address in the system virtual address space. A system space address is shown in Figure 4-7.

The system virtual-address space is defined by the system page table, which is a physically contiguous table of page table entries. The system page table is a page-aligned data structure, located in either virtual or physical memory depending on the processor implementation. Its base address is contained in the system-base register (SBR), shown in Figure 4-8a.

The SBR points to the first PTE in the system page table. In turn, this PTE maps the first page of system space, virtual addresses 80000000 through 800001FF (hex).

The size of the system page table in longwords (that is, the number of PTEs) is contained in the system-length register (SLR), shown in Figure 4-8b. 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. In order to express the values 0 through 2**21 (200000 (hex)) inclusive, the length field in the SLR is 22 bits. SLR values not in the range of 0 to 200000 (hex) inclusive are reserved values and result in UNDEFINED operation.

For all physical address modes, the processor locates the system page table in physical memory and holds a physical address in SBR. The processor uses this function to generate a physical address from a system region virtual address:

```
        SYS_PA = (SBR+4*SVA<29:9>)<PFN>'SVA<8:0>
```

Figures 4-9 and 4-10 illustrate the translation of a system virtual address to a physical address using this function.

In order to make it easier for processors to read aligned blocks of PTEs, software must pad the end of the system page table with invalid PTEs to a 32-byte boundary.

Processor initialization leaves the contents of SBR and SLR UNPREDICTABLE. If part or all of the system page table resides in I/O space, in nonexistent memory, or beyond the end of the physical-address space while memory mapping is enabled, the operation of the processor is UNDEFINED.

The entire system page table must reside in the portion of physical memory that is accessible by virtual address translation when memory management is off. For example, on a processor which implements 34-bit physical address mode (as described in 4.2.1), software must ensure that the system page table resides within the first 512 megabytes of physical memory.

Because the processor may use the value of SBR<n:9> (where n is the most significant bit of SBR used by the processor) to distinguish between system-space address translations of different virtual machines, software on processors that support virtual machines must ensure that SBR<n:9> is unique for each virtual machine. \In effect, if different VMs are mapped by different system page tables, then the page tables have to be at different addresses. The only way of NOT meeting this requirement would be to have VMs with the same page table, but different page table lengths. The requirement has no effect on the values of SBR within virtual machines.\

```
 31                                            9 8               0
+-----------------------------------------------+-----------------+
|       address of system page table            |       MBZ       |
+-----------------------------------------------+-----------------+
```

a. System Base Register (SBR)

```
 31                22 21                                         0
+--------------------+--------------------------------------------+
|        MBZ         | length of system page table in longwords   |
+--------------------+--------------------------------------------+
```

b. System Length Register (SLR)

Figure 4-8 System Page-Table Mapping Registers

```
                            3 3 2
                            1 0 9                 9 8     0
  system-space             +---+-------------------+-------+
  virtual address:         |   |virtual page number| byte  |
                           +---+-------------------+-------+
                               |                   |\       \
                               |    extract VPN,   | \       \
                               |    check length,  |  \       \
                       3      2|2     and add      |   \       \
                       1      3|2                 2|1 0 \       \
                      +--------+-------------------+---+ \       \
| SBR:                |    physical address of SPT base|  \       \
|                     +--------------------------------+   |       |
|                     |                                |   |       |
|                     |                                |   |       |
|                     |              yields            |   |       |
|                     |                                |   |       |
|                     |                               0|   |       |
|                     +--------------------------------+   |       |
|                     |      physical address of SPTE  |   |       |
|                     +--------------------------------+   |       |
                                                           |       |
                                                           |       |
                                       fetch               |       |
                       3    2 2                            |       |
                       1    5 4                       0    |       |
                      +------+-------------------------+   |       |
  SPTE:               |      |    page frame number    |   |       |
                      +------+-------------------------+   |       |
                             |                         |   |       |
                             |                         |   |       |
                             |          merge          |  /       /
                             |3                        | /       /
                             |3                       9|/8    0 /
                             +-------------------------+-------+
  physical address:          |    page frame number    | byte  |
                             +-------------------------+-------+
```

Figure 4-9 System Virtual-Address Translation (25-Bit PFN)

```
                         3 3 2
                         1 0 9                 9 8     0
system-space            +---+-------------------+-------+
virtual address:        |   |virtual page number| byte  |
                        +---+-------------------+-------+
                            |                   |\       \
                            |    extract VPN,   | \       \
                            |    check length,  |  \       \
                    3 3 2  2|2     and add      |   \       \
                    1 0 9  3|2                 2|1 0 \       \
                   +---+----+-------------------+---+ \       \
SBR:               |   |physical address of SPT base|  \       \
                   +---+----------------------------+   |       |
                       |                            |   |       |
                       |                            |   |       |
                       |          yields            |   |       |
                       |2                           |   |       |
                       |9                          0|   |       |
                       +----------------------------+   |       |
                       |  physical address of SPTE  |   |       |
                       +----------------------------+   |       |
                                                        |       |
                                                        |       |
                                    fetch               |       |
                    3        2 2                        |       |
                    1        1 0                   0    |       |
                   +----------+---------------------+   |       |
SPTE:              |          |  page frame number  |   |       |
                   +----------+---------------------+   |       |
                              |                     |   |       |
                              |                     |   |       |
                              |         merge       |  /       /
                              |2                    | /       /
                              |9                   9|/8    0 /
                              +---------------------+-------+
physical address:             |  page frame number  | byte  |
                              +---------------------+-------+
```

| Figure 4-10 System Virtual-Address Translation (21-Bit PFN)

4.6 PROCESS-SPACE ADDRESS TRANSLATION

The process virtual address space is divided into two, equal size, 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. Figure 4-11 illustrates a process virtual address.

The P0 region maps a virtually contiguous area that begins at the smallest address (0) in the process virtual-address 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-address 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 contiguous vector of page table entries. Like the System Page Table, the process page tables may be addressed by either virtual or physical addresses, depending on the processor implementation. (A processor that implements process page tables in physical memory is known as an rtVAX variant. Such processors are described in Chapter 11, Implementation Options.)

There is reason to address process page tables in virtual rather than physical space: a physically addressed process page table that requires more than a page of PTEs (that is, that maps more than 64K bytes of process virtual-address space) requires physically contiguous pages. Such a requirement makes dynamic allocation of process page table space very awkward since a running system tends to fragment storage into page-size areas.

There is also reason to address process page tables in physical memory: 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 by the processor from system space, the system space page containing the process PTE may be marked valid or invalid. If it is marked valid, the processor can read the process space PTE. If the system space page is invalid, a translation-not-valid fault results, and the "PTE reference" bit is set in the fault parameter. This allows the process page tables to be paged.

The operating system must make process page tables accessible to kernel mode, at least. The operation of the processor is UNDEFINED if process space page tables are read-only or no-access. Thus the processor may or may not perform access checking (in kernel mode) when reading a process PTE or updating PTE<M> in a process PTE.

When a process PTE is read from system space, 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 translation-not-valid or length-violation faults. (See section 4.8, "Faults and Parameters".)

If part or all of either process page table is mapped into I/O space or nonexistent memory while memory mapping is enabled, the operation of the processor is UNDEFINED.

4.6.1 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 page-aligned virtual address in the system region that is the base address of the P0PT. Figure 4-12a illustrates the P0BR. The P0LR contains the size of the P0PT in longwords, that is, the number of page table entries. Figure 4-12b illustrates the P0LR. The page table entry addressed by the P0BR maps the first page of the P0 region of the virtual address space, that is, virtual byte address 0.

The virtual page number is contained in bits <29:9> of the virtual address. There could be as many as 2**21 pages in the P0 region. In order to express the values 0 through 2**21 (200000 (hex)) inclusive, the length field in the P0LR is 22 bits. P0LR values not in the range of 0 to 200000 (hex) inclusive are reserved values and result in UNDEFINED operation.

Because P0BR holds a page-aligned virtual address, P0BR<8:0> must be zero (MBZ). In kernel mode, the operation of the processor is UNDEFINED after the execution of an MTPR to P0BR which writes a non-zero value to P0BR<8:0>.

An attempt to load P0BR with a value less than 2**31 or greater than 2**31 + 2**30 - 4 results in a reserved-operand fault in some implementations. Writing P0LR bits <26:24> has no effect. P0LR bits <26:24> read as zero. At processor initialization time, the contents of both P0BR and P0LR are UNPREDICTABLE.

The algorithm to generate a physical address from a P0 region virtual address is as follows:

```
        P0PTE_VA = P0BR+4*PVA<29:9>
        P0PTE_PA = (SBR+4*P0PTE_VA<29:9>)<PFN>'P0PTE_VA<8:0>
        ! Fetch from physical memory
        P0_PA    = (P0PTE_PA)<PFN>'P0_VA<8:0>
```

Figures 4-13a and 4-13b illustrate the translation of a process virtual address to a physical address using this function.

In order to make it easier for processors to read aligned blocks of PTEs, software must pad the end of the P0 page table with invalid PTEs to a 32-byte boundary.

```
 3 3 2
 1 0 9                                 9 8               0
+-+-+------------------------------------+-----------------+
|0|x|         virtual page number        | byte within page|
+-+-+------------------------------------+-----------------+
```

Figure 4-11 Process Virtual-Address Format

```
 3 3 2
 1 0 9                                                   0
+----------------------------------------------------------+
|    page-aligned system virtual address of P0BR page table |
+----------------------------------------------------------+
```

a. P0 Base Register (P0BR)

```
 3         2 2   2 2 2 2
 1         7 6   4 3 2 1                                 0
+----------+-----+---+-------------------------------------+
|   MBZ    | IGN |MBZ|        length of P0PT in longwords  |
+----------+-----+---+-------------------------------------+
```

b. P0 Length Register (P0LR)

```
 3 3 2
 1 0 9                                                   0
+----------------------------------------------------------+
|                 page-aligned virtual address             |
+----------------------------------------------------------+
```

c. P1 Base Register (P1BR)

```
 3 3               2 2
 1 0               2 1                                   0
+-+-----------------+--------------------------------------+
|I|       MBZ       |   2**21 - length of P1PT in longwords |
+-+-----------------+--------------------------------------+
```

d. P1 Length Register (P1LR)

Figure 4-12 Process-Space Mapping Registers

```
                         3 3 2
                         1 0 9                 9 8     0
process-space           +---+-------------------+-------+
virtual address:        |   |virtual page number| byte  |
                        +---+-------------------+-------+
                            |                   |\       \
                            |    extract VPN,   | \       \
                            |    check length,  |  \       \
                     3     2|2     and add      |   \       \
                     1     3|2                 2|1 0 \       \
                    +-------+-------------------+---+ \       \
PxBR:               | virtual address of PxPT base  |  \       \
                    +-------------------------------+   \       |
                    |                               |    \      |
                    |                               |     \     |
                    |             yields            |      \    |
                    |3 3 2                          |       |   |
                    |1 0 9                 9 8     0|       |   |
virtual address     +---+-------------------+-------+       |   |
of PxPTE:           |   |virtual page number| byte  |       |   |
                    +---+-------------------+-------+       |   |
                         fetch using system-space           |   |
                          translation algorithm,            |   |
                   including length and validity checks     |   |
                           3    2 2                         |   |
                           1    5 4                     0   |   |
                          +------+-----------------------+  |   |
PxPTE:                    |      |   page frame number   |  |   |
                          +------+-----------------------+  |   |
                                 |                       |  |   |
                                 |                       |  |   |
                                 |          merge        |  |   |
                                 |3                      | /    |
                                 |3                     9|/8   0|
                                 +-----------------------+------+
physical address:                |   page frame number   | byte |
                                 +-----------------------+------+
```

a. Translation with a 25-Bit PFN

Figure 4-13 Process Virtual-Address Translation

```
                         3 3 2
                         1 0 9                  9 8     0
process-space           +---+-------------------+-------+
virtual address:        |   |virtual page number| byte  |
                        +---+-------------------+-------+
                            |                   |\       \
                            |    extract VPN,   | \       \
                            |    check length,  |  \       \
                     3     2|2     and add      |   \       \
                     1     3|2                 2|1 0 \       \
                    +-------+-------------------+---+ \       \
PxBR:               | virtual address of PxPT base  |  \       \
                    +-------------------------------+   \       |
                    |                               |    \      |
                    |                               |     \     |
                    |             yields            |      \    |
                    |3 3 2                          |       |   |
                    |1 0 9                 9 8     0|       |   |
virtual address     +---+-------------------+-------+       |   |
of PxPTE:           |   |virtual page number| byte  |       |   |
                    +---+-------------------+-------+       |   |
                         fetch using system-space           |   |
                          translation algorithm,            |   |
                   including length and validity checks     |   |
                           3    2 2                         |   |
                           1    1 0                     0   |   |
                          +------+-----------------------+  |   |
PxPTE:                    |      |   page frame number   |  |   |
                          +------+-----------------------+  |   |
                                 |                       |  |   |
                                 |                       |  |   |
                                 |          merge        |  |   |
                                 |2                      | /    |
                                 |9                     9|/8   0|
                                 +-----------------------+------+
physical address:                |   page frame number   | byte |
                                 +-----------------------+------+
```

b. Translation with a 21-Bit PFN

Figure 4-13 Process Virtual-Address Translation (continued)

4.6.2 P1 Region

The P1 region of the address space is mapped by the P1 page table (P1PT). P1PT is defined by the P1 base register (P1BR) and the P1 length register (P1LR). Because P1 space grows toward 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. Figures 4-12c and 4-12d illustrate the P1 base register and P1 length register. Note that P1LR contains the number of nonexistent PTEs. P1BR contains a page-aligned 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.

Because P1BR holds a page-aligned virtual address, P1BR<8:0> must be zero (MBZ). In kernel mode, the operation of the processor is UNDEFINED after the execution of an MTPR to P1BR which writes a non-zero value to P1BR<8:0>. 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.

There could be from 0 to 2**21 nonexistent PTEs in the P1 page table. In order to express the values 0 through 2**21 (200000 (hex)) inclusive, the length field in the P1LR is 22 bits. P1LR values not in the range of 0 to 200000 (hex) inclusive are reserved values and result in UNDEFINED operation. Writing P1LR<31> has no effect. The bit always reads as 0.

At processor initialization time, the contents of both P1BR and P1LR are UNPREDICTABLE.

The algorithm to generate a physical address from a P1 region virtual address is as follows:

```
        P1PTE_VA = P1BR+4*PVA<29:9>
        P1PTE_PA = (SBR+4*P1PTE_VA<29:9>)<PFN>'P1PTE_VA<8:0>
        ! Fetch from physical memory
        P1_PA    = (P1PTE_PA)<PFN>'P1_VA<8:0>
```

Figures 4-13a and 4-13b illustrate the process virtual address to physical address translation.

In order to make it easier for processors to read aligned blocks of PTEs, software must pad the beginning of the P1 page table with invalid PTEs from a 32-byte boundary. That is, the following number of longwords must immediately precede the longword at P1BR + 4*P1LR and contain PTEs with PTE<V> clear: (P1LR modulo 8) longwords.

4.7 TRANSLATION BUFFER

In order to avoid rereading PTEs when repeatedly translating references to the same virtual pages, a scalar or vector processor may include a mechanism to remember successful virtual address translations and page states. Such a mechanism is termed a translation buffer (TB). Any time a page table is active (that is, any time it is addressed by a page table base and length register, and MME=1), the processor may read and cache valid PTEs from that page table in a TB, and may use them for translating addresses. The processor must not cache invalid PTEs. A cached PTE is also referred to as a TB entry. \Besides caching a virtual page's PTE in the TB, a processor may also cache portions of the virtual page in a virtual data cache. In such processors, operations, invalidations, and associations performed on the TB entry must also affect the virtual page's entries in the virtual data cache. Thus, system or process entries in the virtual data cache are considered part of system or process TB state respectively.\

Processors which implement the address space number register, ASN, associate the current value of the ASN register with process-space TB state created for the current process. In this way, such a processor may continue to keep this TB state even after the page table that was used to obtain it becomes inactive and so preserve the TB state until the next time the process runs. (But the processor must not read PTEs from inactive page tables or use such PTEs to translate addresses.) Further, the processor may use this TB state in translating and servicing the references of other processes which share the same value of ASN. \Implementations must ensure that both LDPCTX and MTPR-to-ASN make the new value of the ASN IPR available to TB and virtual cache logic.\ If such a processor supports virtual machines, it also associates system-space TB state with SBR values; specifically, with SBR<n:9>, where n is the most significant bit of SBR used by the processor. (SBR<8:7> are zero, since the system page table is page aligned.)

Software is responsible for ensuring TB consistency by flushing stale PTEs from the TB. Between the times when software modifies a valid PTE in an active page table and when software flushes the old ("stale") value of that PTE from any TBs that may have cached it, processors may use either the old value of the PTE or the new value of the PTE for translating references and for updating the PTE<M> bit. Software must flush the entire TB before enabling memory mapping. Software can flush the entire TB on both the scalar and vector processors by writing zero to the Translation-Buffer Invalidate All (TBIA) internal processor register. Software can flush the entire TB of just the vector processor by writing zero to the Vector Translation-Buffer Invalidate All (VTBIA) internal processor register. Software can also flush the translation of a page mapped by an active page table from the TB of both the scalar and vector processors by writing any address in that page to the Translation-Buffer Invalidate Single (TBIS) internal processor register.

On processors which implement the ASN register, software can flush all process TB state on the VAX scalar processor associated with an address space number by writing the ASN value into the TBIASN internal processor register, or by executing a LDPCTX instruction with PCB<PRVCPU> not equal to the current processor's CPUID. The format of the TBIASN IPR is shown in figure 4-14. Also, software can flush all system-space TB state on the VAX scalar processor associated with a system page table by writing the page table's SBR to TBISYS. TBIASN and TBISYS do not affect the TB state of the vector processor.

To save software the cost of explicit invalidates after software sets PTE<M>, processors that implement modify fault flush the faulting PTE from the TB as part of the process of initiating a modify fault. Note that the TB may reload and cache the old PTE value between the time when modify fault flushes the old value and the time when software updates the PTE in memory. Software that depends on the hardware-provided flush must thus be prepared to take another modify fault on a page after setting the page's PTE<M> bit. The second modify fault will flush the stale PTE from the TB, and the processor cannot load another stale copy. Thus in the worst case, a multiprocessor system will take an initial modify fault and then an additional modify fault on each processor. In practice, even a single repetition is unlikely.

After software changes a valid system PTE that maps any part of a process page table, software must flush the translation for the system page and then flush the translations of all valid process pages so mapped. If software writes such a process page before flushing its translation, the operation of the processor is UNDEFINED.

Because the TB must not store invalid PTEs, software is not required to flush TB entries after changing PTEs that were already invalid.

When software changes the location of a page table, it must flush translations as though it had modified all the PTEs the page table contained. When software reduces the size of a page table, it must flush translations as though it had modified all the PTEs that were part of the page table and are no longer.

Software is also responsible for ensuring that other processors in a multiprocessor don't use stale PTEs and that scalar/vector memory synchronization is performed before PTE modification.

Software can use the following suggested protocol to properly coordinate invalidation:

1. Acquire exclusive access to the PTE: Prevent other processes, interrupt service routines, and software running on other processors from writing either the PTE or the page that the PTE maps. (Software must refrain from writing the page in order to prevent a processor from writing to the PTE to set PTE<M>.)

2. Ensure that scalar/vector memory synchronization is performed by each processor which has a vector processor present. (This ensures that vector processor loads and stores are no longer executing and that all stores made by the vector

processor are visible to its associated scalar processor.)

3. Update the PTE.

4. Flush the old value of the PTE from any TBs that may have cached it: If the PTE was invalid, then it couldn't have been cached in any TBs, so software needn't flush it. If a processor's TB has already been flushed since the last time the page table was active on that processor, there's no need to flush it again. But if the PTE was valid and part of a page table that was active on a particular processor, software must assume that the PTE may be cached in that processor's TB -- even if software on that processor has made no explicit reference to the page -- so software must flush the PTE from that processor's TB.

5. Release exclusive access to the PTE.

Other, more efficient, protocols are possible in restricted cases.

When software writes a virtual address to the Translation-Buffer Check (TBCHK) internal processor register, the processor may set the condition code V-bit if the TB contains a valid translation of that address, and clears the V-bit if the TB does not. (PSL<N>, PSL<Z>, and PSL<C> are UNPREDICTABLE.) The results are useful for heuristic paging algorithms. Because the translation buffer can flush translations at any time, and can load PTEs from active page tables at any time, the results may be out of date by the time they are read, and so may not be useful for other purposes.

The TBIS, TBIA, TBCHK, TBIASN, TBISYS processor registers are write only. The operation of MFPR from any of these registers is UNDEFINED.

Before enabling memory management, software must flush the entire translation buffer. If software does not follow this rule, the operation of the processor becomes UNDEFINED when memory management is enabled.

Note that vector processor memory access instructions must not be used to read or write active page tables.

```
 3
 1                                              8 7                0
+----------------------------------------------+------------------+
|        Address Space Number                  |     ignored      |
+----------------------------------------------+------------------+
```

Figure 4-14 TB Invalidate (based on) Address Space Number (TBIASN)

4.8 FAULTS AND PARAMETERS

Three types of faults are associated with memory mapping and protection. 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. A modify fault is taken when a write or modify reference is attempted to a page that has not previously been modified (PTE<26>=0). Note that these three faults have distinct vectors in the system control block. If more than one type of fault could occur on a reference to a single page, then access-control-violation fault takes precedence over both translation-not-valid fault and modify fault. 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. The fault software does not have to compute the length check because a "length violation" indication is stored in the memory management fault stack frame, illustrated in Figure 4-15.

The same parameters are stored for all memory-management faults, including those of the vector processor. 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 (except on an rtVAX processor). 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 fields of the second parameter are described in Table 4-4.

```
 3
 1                                                 6 5 4 3 2 1 0
+---------------------------------------------------+-+-+-+-+-+-+-+
|                                                   |V|V|V| | | |
|                           0                       |A|I|A|M|P|L| :(SP)
|                                                   |S|O|L| | | |
+---------------------------------------------------+-+-+-+-+-+-+-+
|           some virtual address in the faulting page           |
+---------------------------------------------------------------+
|                      PC at time fault taken                   |
+---------------------------------------------------------------+
|                      PSL at time fault taken                  |

+---------------------------------------------------------------+
```

Figure 4-15 Memory-Management-Fault Stack Frame

Table 4-4: Fields of the Memory-Management-Fault Parameter

Name	Extent	Meaning
MBZ	<31:6>	Reserved to DIGITAL.
vector asynchronous MME	<5>	(VAS) Set to 1 to indicate that a vector processor memory management exception has occurred when the asynchronous memory management scheme is implemented.
vector I/O reference	<4>	(VIO) Set to 1 on some vector implementations to indicate that an access control violation has occurred due to a vector instruction reference to I/O space.
vector alignment	<3>	(VAL) Set to 1 when a vector access control violation has occurred due to a vector element not being properly aligned in memory.
modify or write intent	<2>	Set to 1 to indicate that the instruction's intended access was write or modify. This bit is 0 if the instruction's intended access was read. This bit is always set for a modify fault.
PTE reference	<1>	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 translation-not-valid or length-violation faults. This bit is always clear for a modify fault, and on an rtVAX.
length violation	<0>	Set to 1 to indicate that an access-control-violation fault was the result of a length violation rather than a protection violation. This bit is always 0 for a translation-not-valid fault and for a modify fault.

4.9 PRIVILEGED SERVICES AND ARGUMENT VALIDATION

This section lists the instructions allowing access mode change, and describes two instructions that allow privileged services to check addresses passed as parameters.

4.9.1 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 executive
CHMS	change mode to supervisor
CHMU	change mode to user

These instructions provide the normal mechanism for less privileged code to call more privileged code; the instructions are described in detail in Chapter 5, Interrupts and Exceptions. 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.

4.9.2 Validating Address Arguments

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. The PROBE instructions do this verification.

```
        PROBEx          PROBE ACCESSIBILITY

        verify that arguments can be accessed
```

Format:

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

Operation:

```
        probe_mode <- MAXU (mode<1:0>, PSL<PRV_MOD>);
        flag <- {accessibility of base} AND
                {accessibility of {base+ZEXT(len)-1}}
                using probe_mode;

        ! Test whether the pages are valid
        if PSL<VM> EQLU 1 AND flag EQLU {accessible} AND
                {{PTE<V> for base EQLU 0} OR
                 {PTE<V> for {base+ZEXT(len)-1} EQLU 0}}
                   then {VM-emulation trap};
                ! Mode, length, and base address are pushed in
                ! exception frame.
```

Condition Codes:

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

Exceptions:

```
        translation not valid
        modify
        VM emulation
```

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<PRV_MOD>.

If the probe is executed by a processor running in VM mode, and the pages are accessible to the probe mode, and one or both of the mapping PTEs are not valid, then a VM-emulation trap is taken.

Notes:

1. If the valid bit of the examined PTE is set, and write access is allowed, it is UNPREDICTABLE whether the modify bit of the examined PTE is set by a PROBEW. If the valid bit is clear or if write access is not allowed, the modify bit is not changed.

2. If PSL<VM> is set then the valid bit of the examined PTE is used to determine whether to take a VM-emulation trap.

3. If PSL<VM> is set, and if the modify bit of the examined PTE is set by a PROBEW, and if the modify bit of the examined PTE changes state from clear to set, then the processor takes a modify fault.

4. If the probe is executed by a processor running in VM mode, and the first and last bytes specified by the base address and the zero-extended length are in different pages, and the PTE mapping one page is invalid, and the PTE mapping the other page is valid but the page is not accessible to the probe mode, then the instruction will either initiate a VM-emulation trap or complete indicating inaccessibility (PSL<Z> set). Which result occurs is UNPREDICTABLE and left to the discretion of the implementation.

5. Except for the above notes, the processor ignores the valid bit of the PTE mapping the probed address.

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

7. During the probe of a process virtual address, on a non-rtVAX processor, if the valid bit of the system PTE is 0 then a translation-not-valid fault occurs. This allows for the demand paging of the process page tables. On an rtVAX, the process-space page tables are located in physical memory, and this fault cannot occur. (See section 11.2.4 for more information on rtVAX memory management.)

8. An object one byte long is the smallest that can be probed. With a length of zero, the PROBE instructions test the accessibility of two bytes (base and base-1).

9. If memory management is disabled, all memory is accessible, and probing nonexistent memory gives UNPREDICTABLE results.

Example:

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

Flows:

The following describes the operational flow 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 its residency. However, probing a process address may cause a page fault in the system address space on the per-process page tables.

\ These flows assume that an invalid PTE will never get into the TB.\

1. Look up the virtual address in the TB. 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 per-process virtual address, go to step 4. Form physical address of PTE, fetch the PTE, and use the protection field to determine the accessibility. If not in VM mode, EXIT. Determine whether to take a VM-emulation trap based on accessibility to probe mode and validity of the PTE. If a VM-emulation trap is indicated, take it; otherwise EXIT.

4. If per-process virtual address on an rtVAX, form physical address of PTE, fetch the PTE, use the protection field to

determine the accessibility, and EXIT.

5. If process-space virtual address on a non-rtVAX processor, must do a virtual memory reference for the PTE.

   a. Look up the virtual address of the PTE in the TB. If not found, go to step b, below. Form the physical address of the PTE, fetch the PTE, and use the protection field to determine the accessibility. If not in VM mode, EXIT. Determine whether to take a VM-emulation trap based on accessibility to probe mode and validity of the PTE. If a VM-emulation trap is indicated, take it; otherwise EXIT.

   b. If the virtual address of the PTE is not in the TB, check the system virtual address of the PTE for length violation. If length violation, then return No Access and EXIT. \This length violation is clearly an operating-system error and should never happen.\

   c. Read the SPTE for the system-space page containing the per-process PTE.

   d. If the valid bit in the SPTE is 0, then take a translation-not-valid fault and EXIT. This case allows for the demand paging of per-process page tables.

   e. Finally, calculate the physical address of the per-process PTE from the PFN field of the SPTE (see the section "System Space Address Translation" in this chapter), fetch the per-process PTE, and use the protection field to determine the accessibility. If not in VM mode, EXIT. Determine whether to take a VM-emulation trap based on accessibility to probe mode and validity of the PTE. If a VM-emulation trap is indicated, take it; otherwise EXIT.

Change History:

Revision J. Rich Brunner, Tim Leonard, December 1989

- o Added descriptions where appropriate of Vector MM facilities including PTE update, TB considerations, and MM Fault Parameter.
- o Heavily reworked memory management disabled section to include new 32-bit Physical address modes.
- o In TB section: put requirement that TB flush needed before MAPEN back into regular text. It had been put into an implementation note. Reworked implementation note.
- o Clarified that PxBR<8:0> is MBZ.
- o Added ASN IPR which revised ECO 103 additions. Value of ASN IPR not PCBB<29:7> (which was added by ECO 103) and is associated with per-process TB state.
- o Simplified description of SPTEP algorithm. Later removed SPTEP completely from the architecture.
- o Legal values for SLR & PxLR are from 0-200000 (hex) inclusive
- o Fix formula for padding P1 page table to 32-byte boundary.
- o Add ECO 103 -- Process IDs.
- o Fix several minor problems with SPTEP descriptions.
- o Clarify only PPTE<M>=0 causes a modify fault.

Revision H. Tim Leonard, January 1987.

- o Support virtual machines.
- o Make modify fault flush the faulting TB entry.
- o Require page tables to be page aligned, so that PTEs can be more easily read in blocks.
- o Require software to pad page-table ends up to 32-byte boundaries.
- o Extend the physical address; allow SCBB, PCBB, and SBR to contain virtual addresses; add SPTEP; add modify fault.

Revision F. Al Thomas, November, 1986.

- o Condition Codes are UNPREDICTABLE after MxPR.
- o Change references to rtVAX subset to rtVAX processor.
- o Enabling Memory Management need not flush the instruction buffer.

Revision E. Al Thomas, September 1986.

- o Add text describing rtVAX memory management.
- o Clarify the description of length violation.

Revision D1. Tim Leonard, January 1986.

- o PROBEW mustn't set PTE<M> if write access is not allowed.

Revision D. Tim Leonard, March 1985.

- o Change the revision number to correspond to DEC Standard 032 rev number.
- o When memory management is disabled, existent memory is accessible, and PROBEing nonexistent memory is UNPREDICTABLE.
- o When memory management is disabled, if the IB prefetches into nonexistent memory, the operation of the processor is UNDEFINED.
- o Allow processors to read process PTEs without access checks. No longer does a no-access SPTE take the place of 128

no-access PxPTEs.
- o Add note to PROBE on length of zero.

Revision 6. Tim Leonard, 26 July 1982.
- o Clarify description of PTE<M>.
- o Add note to PROBE on the setting of PTE<M>.
- o Add constraint on software using page tables.
- o Add description of TBCHK IPR.
- o Rework PROBE and implied probe write.

Revision 5, DR32 additions. Tom Eggers, 8 February 1980.
- o Reorganize to match Handbook (Volume II).
- o Add explanatory material to match Handbook.
- o Delete ISSUES section, except for "Access Across a Page Boundary."
- o Reword introduction to address translation.
- o Added instructions to read and write each IPR.
- o Remove numeric references to other chapters or sections. Replace with chapter or section titles.
- o Add memory management architecture for DR32. New PTE formats. Reserved bits changed to Software bits.
- o A longword write, when aligned, is an "atomic" operation. This is added for PTE changes during multiprocessing.
- o Simplify rule for when TB entry must be invalidated. State explicitly that TB holds no invalid PTEs.
- o Clarify actions when MME=0.
- o Clarify clearing TB before setting MME=1.
- o Clarify when PTE<M> is set.

Revision 4, document registers. Peter Conklin and Ted Taylor. 30 July 1978.
- o PTE<24:23> now for software use; <25> and <22:21> are untested MBZ.
- o Include boot initial state.
- o Correct picture formats.
- o Add descriptions of translation buffer, unmapped, MAPEN, TBIA, and TBIS.
- o Add picture of stack on fault.
- o Add example of translation.
- o Cleanup the description of PROBE.
- o P0LR<26:24> and P1LR<31> ignored on MTPR.
- o PROBE uses only mode<1:0>; ignores rest of operand.
- o Remove redundant, confusing pseudo flows.
- o Remove some argumentative justifications.
- o Specify which registers are read/write.
- o SLR is checked on process page table references.
- o Clarify address on fault.
- o M not necessarily maintained in SP of process page tables.
- o Add software mnemonics.
- o SWU bits renamed to OWN and reserved for owning access mode.
- o M update is not interlocked in multiprocessor.
- o Add comments on accesses across page boundaries.

Revision 3, results of April Task Force review. Peter Lipman, 4 June 1976.

- o Eliminate the concept of segments, moving protection to the page level.
- o Streamline the virtual to physical address translation by eliminating references to the PCB and Internal Segment Table.
- o Remove access level from virtual address.
- o Eliminate Pointer Segments and Stack Segments.
- o Add mode operand to PROBE.
- o Assign codes to protection field.
- o Change sense of condition codes on PROBE.
- o PROBE length is unsigned.
- o Protection field is valid even if V=0.
- o P1BR specifies lowest (unused) address in P1 space.
- o Modify bit moved to Page Table Entry.
- o Memory Mapping Enable Bit.

Revision 2, ECOs 1 through 18. Tom Hastings, March 1976.

- o Remove execute protection.
- o Added flows for translation.

Revision 1. Tom Hastings, October 1975.

CHAPTER 5

INTERRUPTS AND EXCEPTIONS

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 an event is termed an exception. Both the scalar and vector processors generate exceptions; however, they differ in the mechanism they use to notify the currently executing process. This chapter discusses exceptions generated by the scalar processor. Section 14.2 discusses exceptions generated by the vector processor.

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." 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).

The processor has 31 interrupt priority levels: 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, #PR$_SIRR. (See Chapter 8, and the section "Software 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 if the system has a UNIBUS interconnect; UNIBUS levels BR4 to BR7 correspond to VAX interrupt levels 14 to 17 (hex). Interrupt levels 18 to 1F (hex) are for use by urgent conditions, serious errors, and powerfail.

The processor arbitrates interrupt requests according to priority. Only when the priority of an interrupt request is higher than the processor's current IPL (stored in PSL<20:16>) will the processor raise its 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 interrupt service routine.

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, #PR$_IPL where src contains the new priority desired. However, a processor cannot disable interrupts on other processors. Furthermore, the priority level of one processor does not affect the priority level of the other processors. Thus in multiprocessor systems, interrupt priority levels cannot be used to synchronize access to shared resources. Even the various urgent interrupts including those exceptions that run at IPL 1F (hex) do so on only one processor. Consequently, special software action is required to stop other processors in a multiprocessor system.

Underlying the VAX 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 for software to operate deterministically, 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 must allow the interrupt to occur, if processor priority permits, before executing the apparently subsequent instruction.

Most exception service routines execute at IPL 0 in response to exception conditions caused by the software. A variation from this is serious system failures, which raise IPL to the highest level (1F, hex) to minimize processor interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions; however, nested exceptions can occur.

A trap is an exception 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 trap conditions by using the BISPSW and BICPSW instructions described in Chapter 3.

A fault is an exception 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. After an instruction faults, the PC saved on the stack points to the instruction that faulted. 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 that occurs during an instruction. An abort leaves the value of registers and memory UNPREDICTABLE such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone. After an instruction aborts, the PC saved on the stack points to the opcode of the aborted instruction. The following are UNPREDICTABLE:

- Destination operands (including implied operands, such as the top of the stack in an JSB instruction)

- Registers modified by operand specifier evaluation (including specifiers for implied operands)

- PTE<M> in a PTE that maps a write-access or modify-access operand, where the implementation sets PTE<M> rather than taking modify faults, and write access to the operand was allowed, and the operand was not written, and PTE<M> was clear at the beginning of the instruction

- Condition codes

- PSL<FPD>

- PSL<TP>, if PSL<T> was set at the beginning of the instruction

Except where otherwise noted in the description of the abort, the rest of the PSL, other registers, and memory are unaffected. Instructions that abort are constrained only by memory protection.

Generally, exceptions and interrupts are very similar. When either is initiated, both the processor status longword and the program counter are pushed onto the stack. There are, however, several important differences:

- An exception condition is caused by the execution of the current instruction, whereas an interrupt is caused by some activity in the computing system that may be independent of the current instruction.

- An exception condition is usually serviced in the context of the process that produced the exception condition, whereas an interrupt is serviced independently from the currently running process.

- The IPL of the processor is usually not changed when the processor initiates an exception, whereas the IPL is always raised when an interrupt is initiated.

- Exception service routines usually execute on a per-process stack, whereas interrupt service routines normally execute on a per-CPU stack.

- Enabled exceptions are always initiated immediately, no matter what the processor IPL is; whereas interrupts are held off until the processor IPL drops below the IPL of the requesting interrupt.

- Most exceptions cannot 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, the only exception condition 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.

- 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.

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 an uncorrectable memory-read error. In this case, the interrupt request associated with the 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. The interrupt is still pending, however, and will be taken when the priority is lowered.

5.1 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 PC and the PSL on the stack. The PC and PSL are later restored with the Return from Exception or Interrupt (REI) instruction. Any other status required to correctly resume an interruptible instruction is stored in the general registers. 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, as on the stack.

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 6. Other processor status is changed even less frequently; refer to the privileged register descriptions in Chapter 8.

5.2 RESTARTABILITY

The VAX 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, and no memory operands can be written. For some complex or iterative instructions, described in Chapter 3, intermediate results are stored in the general registers, and memory contents may have been altered. For most instructions with only a single modified or written operand, this implies special processing only when a multiple-byte operand spans a protection boundary making it necessary to test accessibility of both parts of the operand. If the instruction has more than one operand to be written, every operand must be probed before any are written. (Writes to operands in memory above the top of the stack are not constrained by this rule. See section 5.7.5.)

Instructions that store intermediate results in the general registers must not compromise system integrity. Therefore they must ensure that any addresses stored or used are virtual addresses, subject to protection checking. In addition, any state information stored or used cannot result in a non-interruptible 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. See Chapter 7, Memory and I/O.

The setting of PTE<M> is specifically excluded from the constraint that memory not be altered until the instruction can be completed.

5.3 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, interrupts are initiated when the instruction state can be completely contained in the registers, PSL and PC; saving additional instruction state in memory is thus avoided.

The following events cause interrupts:

- Device completion (IPL 10 through 17, hex)
- Device error (IPL 10 through 17, hex)
- Device alert (IPL 10 through 17, hex)
- Device memory error (IPL 10 through 17, hex)
- Console terminal transmit and receive (IPL 14, hex)
- Interval timer (implementation dependent, IPL 16 or 18, hex)
- Recovered memory or bus or processor errors (implementation dependent, IPL 18 through 1D, hex)
- Bus errors, processor errors, or uncorrectable memory errors (implementation dependent, IPL 18 through 1D, hex)
- Powerfail (IPL 1E, hex)
- Software interrupt invoked by MTPR src, #PR$_SIRR (IPL 01 through 0F, hex)
- AST delivery when REI restores a PSL with mode greater than or equal to ASTLVL (see Chapter 6) (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.

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.

5.3.1 Urgent Interrupts

The processor provides eight priority levels (18 through 1F, hex) for use by urgent conditions including serious errors and powerfail. Some implementations may not use all eight priority levels. 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 powerfail. Interrupt level 1F (hex) is reserved for those exceptions that must lock out all processing until the condition has been handled. This includes the hardware and software "disasters" (machine-check and kernel-stack-not-valid aborts). It might also be used to allow a kernel-mode debugger to gain control on any exception.

5.3.2 Device Interrupts

The processor provides eight priority levels (10 through 17, hex) for use by peripheral devices. Some implementations may not implement all eight levels of interrupts. The minimal implementation is levels 14 through 17 (hex) that correspond to the UNIBUS levels BR4 through BR7 if the system has a UNIBUS interconnect.

5.3.3 Software Interrupts

The processor provides 15 interrupt levels (1 through 0F, hex) for use by software. Pending software interrupts are recorded in the software-interrupt-summary register (SISR), as shown in Figure 5-1. The SISR contains ones in the bit positions corresponding to levels at which software interrupts are pending. When the processor initiates a software interrupt, the corresponding bit in SISR is cleared. At no time can SISR contain ones in bits corresponding to levels higher than the current processor IPL, since the processor would already have taken the requested interrupts.

At processor initialization, SISR is cleared. The mechanism for accessing it follows:

MFPR #PR$_SISR, dst   Reads the software interrupt summary register.

MTPR src, #PR$_SISR   Loads it, but this is not the normal way of making software interrupt requests. It is useful, for example, for clearing the software interrupt system and for reloading its state during powerfail recovery.

```
 31                              16 15                          1 0
+----------------------------------+-----------------------------+-+
|                                  | pending software interrupts |M|
|              MBZ                 |                             |B|
|                                  |F E D C B A 9 8 7 6 5 4 3 2 1|Z|
+----------------------------------+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5-1 Software-Interrupt Summary Register

```
 31                                                    4 3       0
+-------------------------------------------------------+---------+
|                        ignored                        | request |
+-------------------------------------------------------+---------+
```

Figure 5-2 Software-Interrupt Request Register

```
 31                                                  5 4          0
+---------------------------------------------------+-------------+
|                 ignored; returns 0                 |PSL<20:16>  |
+---------------------------------------------------+-------------+
```

Figure 5-3 Interrupt-Priority-Level Register

Table 5-1: An Example of Interrupt Processing

	State After Event		
Event	IPL (hex)	SISR (hex)	Stacked PSL<IPL>
Initial state:	5	00	0
Execute MTPR #8, #PR$_IPL:	8	00	0
Execute MTPR #3, #PR$_SIRR:	8	08	0
Execute MTPR #7, #PR$_SIRR:	8	88	0
Execute MTPR #9, #PR$_SIRR (interrupts at once):	9	88	8,0
Device interrupt at IPL 20 (decimal):	14	88	9,8,0
Device interrupt service routine executes REI:	9	88	8,0
IPL 9 service routine executes REI:	8	88	0
Execute MTPR #5, #PR$_IPL: *	7	08	5,0
IPL 7 service routine executes REI:	5	08	0
Initial IPL 5 service routine executes REI: *	3	00	0
IPL 3 service routine executes REI:	0	00	--

* This operation lowers IPL below that of an outstanding software interrupt request. The software interrupt occurs at once.

The software-interrupt-request register (SIRR) is a write-only, 4-bit, privileged register used for creating software interrupt requests. SIRR is shown in Figure 5-2.

Executing MTPR src, #PR$_SIRR requests an interrupt at the level specified by src<3:0>. Once a software interrupt request is made, it will be cleared by the hardware when the interrupt is taken. If src<3:0> is greater than the current IPL, the interrupt occurs before execution of the following instruction. If src<3:0> is less than or equal to the current IPL, the interrupt will be deferred until IPL is lowered to less than src<3:0> and there is no higher interrupt level pending. This lowering of IPL is by either REI or by MTPR src, #PR$_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. The service routine, therefore, 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, #PR$_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.

5.3.4 Interrupt Priority Level Register

Writing to the IPL register with the MTPR instruction will load the processor priority field in the PSL; that is, PSL<20:16> is loaded from IPL<4:0>. Reading from the IPL register with the MFPR instruction will read the processor priority field from the PSL. On writing the IPL register, bits <31:5> are ignored; on reading the IPL register, bits <31:5> are returned 0. The IPL register is shown in Figure 5-3. At processor initialization, IPL is set to 1F (hex).

Interrupt service routines must follow the discipline of not lowering IPL below their initial level. If a service routine lowers IPL below its initial level, an interrupt could occur at an intermediate level, which will cause the stack nesting to be improper, which will result in REI faulting. If IPL is lowered to zero when the processor is running on the interrupt stack, the operation of the processor is UNDEFINED. Table 5-1 is an example of interrupt processing.

```
+-----------------------------------------------------------------+
|                            type code                            | :(SP)
+-----------------------------------------------------------------+
|               PC of next instruction to execute                 |
+-----------------------------------------------------------------+
|                               PSL                               |
+-----------------------------------------------------------------+
```

Figure 5-4 Arithmetic-Exception Stack Frame

Table 5-2: Arithmetic-Exception Type Codes

Exception Type	Mnemonic	Decimal	Hex
Traps			
integer overflow	SS$_INTOVF	1	1
integer divide-by-zero	SS$_INTDIV	2	2
decimal divide-by-zero	SS$_FLTDIV	4	4
decimal overflow	SS$_DECOVF	6	6
subscript range	SS$_SUBRNG	7	7
Faults			
floating overflow	SS$_FLTOVF_F	8	8
floating divide-by-zero	SS$_FLTDIV_F	9	9
floating underflow	SS$_FLTUND_F	10	A

Table 5-3: Compatibility-Mode Exception Type Codes

Exception Type	Code
Faults	
reserved opcode	0
BPT instruction	1
IOT instruction	2
EMT instruction	3
TRAP instruction	4
illegal instruction	5
Aborts	
odd address	6

5.4 EXCEPTIONS

Exceptions can be grouped into six classes:

- Arithmetic exceptions
- Memory management exceptions
- Exceptions detected during operand reference
- Exceptions occurring as a consequence of an instruction
- Tracing
- Serious system failures

5.4.1 Arithmetic Exceptions

Arithmetic exceptions may occur as the result of an arithmetic or conversion operation. These exceptions 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, as shown in Figure 5-4. Table 5-2 lists the arithmetic-exception type codes.

Integer Overflow Trap -- An integer overflow trap is an exception that indicates that the last instruction executed had an integer overflow setting PSL<V> and that integer overflow was enabled (PSL<IV> was set). The result stored is the low-order part of the correct result. PSL<N> and PSL<Z> are set according to the stored result. The type code pushed on the stack is 1 (SS$_INTOVF).

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 PSL<V> is set. The type code pushed on the stack is 2 (SS$_INTDIV).

Divide-By-Zero Trap -- 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 R5, and condition codes are UNPREDICTABLE. The zero divisor can be either +0 or -0. The type code pushed on the stack is 4 (SS$_FLTDIV).

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 (PSL<DV> was set). The PSL<V> condition code is always set. Refer to the individual instruction descriptions in Chapter 3 for the value of the result and of the condition codes. The type code pushed on the stack is 6 (SS$_DECOVF).

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 the indexout operand, and the condition codes are set as if the subscript were within range. The type code pushed on the stack is 7 (SS$_SUBRNG).

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 PSL<FPD> set. The type code pushed on the stack is 8 (SS$_FLTOVF_F).

Floating Divide-By-Zero 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 (SS$_FLTDIV_F).

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 (PSL<FU> was 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 PSL<FPD> set. The type code pushed on the stack is 10 (SS$_FLTUND_F).

Floating overflow, floating underflow, and floating divide by zero were originally implemented as traps on the VAX-11/780, and had type codes 3, 4, and 5, respectively. The architecture was later modified to include only floating-point faults, and all VAX-11/780s were upgraded.

5.4.2 Memory Management Exceptions

A memory management exception can be an access-control-violation fault, a translation-not-valid fault, or a modify fault. See Chapter 4, Memory Management, for a description of the information pushed on the stack as parameters by the memory-management exceptions.

Note that in a reference to a single page, if two or more memory-management exceptions are indicated, access-control-violation takes precedence over both translation-not-valid and modify fault.

Access-Control-Violation Fault -- An access-control-violation fault is an exception indicating that the process attempted a reference not allowed at the current access mode. Software may restart the process after changing the address translation information.

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 entry was not set. Software may restart the process after making the page valid.

Modify Fault -- A modify fault is an exception indicating that the process attempted a write-access or modify-access reference to a page for which the modify bit in the page-table entry was not set. The processor flushes the faulting PTE from the TB as part of the process of initiating a modify fault. Software may restart the process after setting PTE<M>. Not all processors take modify faults; some implementations set PTE<M> and continue.

5.4.3 Exceptions Detected During Operand Reference

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.

See Chapter 2 for details of reserved addressing modes and for combinations of addressing modes and registers that cause UNPREDICTABLE results.

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 saved PC to point to the opcode. The exception service routine may determine the type of operand by examining the opcode using the saved 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 labeled as aborts rather than faults. The saved PC is always restored properly unless the instruction attempted to modify it in a manner that results in UNPREDICTABLE results.

5.4.4 Exceptions Occurring as the Consequence of an Instruction

Reserved-Instruction Fault -- A reserved-instruction fault occurs when the processor encounters an opcode that is reserved for future use by DIGITAL. No parameters are pushed. Opcode FFFF (hex) is reserved for all time.

Privileged-Instruction Fault -- A privileged-instruction fault occurs when the processor encounters an instruction that requires higher privileges than the current processor mode. No parameters are pushed.

Privileged-instruction fault and reserved-instruction fault use the same SCB vector, and cannot be distinguished without examining the faulting instruction.

An Opcode-Reserved-To-Customers Fault -- An opcode-reserved-to-customers fault is an exception that occurs when an opcode reserved to customers is executed. The operation is identical to the reserved-instruction fault except that the event is caused by a different set of opcodes, and faults through a different vector. All opcodes reserved to customers start with FC (hex), which is the XFC instruction. If the special instruction needs to generate a unique exception, one of the reserved-to-customer vectors should be used. An example might be an unrecognized second byte of the instruction.

The XFC fault is intended primarily for use with writable control store to implement installation-dependent instructions. The method used to enable and disable the handling of an XFC fault in user-written microcode is implementation-dependent. Some implementations may transfer control to microcode without checking bits <1:0> of the exception vector.

Instruction-Emulation Exceptions -- When a processor executes a string instruction that is omitted from its instruction set, an emulation exception results. An emulation exception can occur through either of two SCB vectors, depending on whether or not PSL<FPD> was set at the beginning of the instruction. If PSL<FPD> is clear, an instruction-emulation trap occurs through the SCB vector at offset C8 (hex), and a instruction-emulation trap frame is pushed onto the current stack. If PSL<FPD> is set, a suspended-emulation fault occurs through the SCB vector at offset CC (hex); and PC and PSL are pushed onto the current stack.

Before any information is pushed on the stack, the stack is probed. If the probe faults, the instruction then faults, else the exception continues as described above.

The emulation exception handler runs in the mode of the emulated instruction, on the same stack, and at the same IPL. The exception parameters are pushed onto the current stack. See Chapter 11 for details of instruction emulation and the emulation exceptions.

VM-Emulation Trap -- A VM-emulation trap occurs when a virtual machine executes a privileged instruction in kernel mode, or when a virtual machine executes a change-mode or REI instruction, or in some cases when a virtual machine executes a PROBEx instruction. (See Chapter 12, Virtual Machines.) The processor initiates the trap to transfer control to the real-machine operating system. The instructions that can cause a VM-emulation trap are HALT, PROBEx, REI, CHMx, LDPCTX, SVPCTX, MTPR, MFPR, PROBEVMW, and PROBEVMR.

Compatibility-Mode Exceptions -- 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 indicating the exception type. The stack frame is the same as that for arithmetic exceptions, shown in Figure 5-4. The compatibility mode exception type codes are shown in Table 5-3.

All other exceptions in compatibility mode occur to the regular native-mode vector; for example, access-control-violation fault, translation-not-valid fault, and machine-check abort. See Chapter 9, PDP-11 Compatibility Mode.

Change-Mode Trap -- A change-mode trap is an exception that occurs when one of the change-mode instructions (CHMK, CHME, CHMS, CHMU) is executed. The instruction operand is pushed on the exception stack, as shown in Figure 5-5. See the description of the change-mode instructions for details.

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 instruction completes, a trace exception will occur (see the section on tracing). At this point, the tracing program can again re-insert the BPT instruction, restore PSL<T> to its original state (usually clear), and resume. Note that if both tracing and breakpointing are in progress (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.

5.4.5 Trace Fault

A trace is an exception that occurs between instructions when trace is enabled. Tracing is used for tracing programs, for performance evaluation, or for 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 as follows:

1. At the beginning of an instruction, if PSL<TP> is set, then a trace fault is taken after clearing PSL<TP>.

2. PSL<TP> is loaded with the value of PSL<T>.

3. If the instruction faults or an interrupt is serviced, PSL<TP> is cleared before the PSL is saved on the stack. The saved 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 saved on the stack.

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 saved on the stack.

The routine entered by a CHMx is not traced because CHMx clears T and TP in the new PSL. However, if PSL<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 PSL<T> was set when the REI was executed or if TP in the saved PSL is set; in both cases, PSL<TP> is set after the REI. Note that a trace fault that occurs for an instruction following an REI that sets PSL<TP> will be taken with the new PSL. Thus, special care must be taken if exception or interrupt routines are traced. If PSL<T> is set by a BISPSW instruction, trace faults begin with the second instruction after the BISPSW.

In addition, the CALLS and CALLG 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, PSL<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.

Table 5-4 shows the operation of tracing during execution of ordinary instructions, instructions that have special effects on tracing, and other system events that effect tracing.

Table 5-4: State Changes During Tracing

Instruction or Instruction Sequence	Original PSL<T>	Final PSL<T>	Final PSL<TP>	Comments
Ordinary instruction	0	0	0	
	1	1	1	
CHMx, through the REI that ends the service routine	0	0	0	
	1	1	1	
Interrupt or exception, through the REI that ends the service routine	0	0	0	
	1	1	1	
CALLS or CALLG	0	0	0	
	1	1	1	(pushed PSW<T> clear)
RET	0	*	0	
	1	*	1	
CHMx	0	0	0	(pushed PSL<TP> clear)
	1	0	0	(pushed PSL<TP> set)
REI, when the stacked PSL<TP> is clear	0	*	0	
	1	*	1	
REI, when the stacked PSL<TP> is set	0	*	1	
	1	*	1	
BISPSW that sets PSW<T>	0	1	0	
	1	1	1	
BICPSW that clears PSW<T>	0	0	0	
	1	0	1	
interrupt or exception	0	0	0	(pushed PSL<TP> clear)
	1	0	0	(pushed PSL<TP> depends on above description)

* Depends on PSW<T> popped from the stack

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 T on in the PSL that will be restored. This defends against programs clearing PSL<T> with RET, REI, or BICPSW.

2. The trace handler should never examine or alter PSL<TP> when continuing tracing. The hardware flows ensure that this bit is maintained correctly to continue tracing.

3. When tracing is to be ended, both PSL<T> and PSL<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 CALLS or CALLG 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 PSL<T> when turning on and off trace. They also would have to simulate traced code that alters or reads PSL<T>.

5.4.6 Serious System Failures

Kernel-stack-not-valid abort is an exception that indicates that a memory-management problem occurred 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 operating system 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 of the original exception. IPL is raised to 1F (hex). If the exception vector <1:0> is not 1, the operation of the processor is UNDEFINED.

\The preferred implementation is to take kernel-stack-not-valid abort on the interrupt stack at IPL 1F (hex) even if the exception vector <1:0> is 0.\

Software may abort the process without aborting the system. Because of the lost information, however, the process cannot be continued. If a memory-management problem occurs on a normal instruction reference to the kernel stack (as with CHMx or REI), the normal memory-management fault is initiated.

An interrupt-stack-not-valid halt results when a memory-management problem or memory error occurs while the processor is pushing information onto the interrupt stack during the initiation of an exception or interrupt. No further interrupt requests are acknowledged on the processor. The processor leaves the PC, the PSL, and the reason for the halt in registers so that they are available to a debugger, to the normal bootstrap routine, or to an optional watch-dog bootstrap routine. A watch-dog bootstrap can cause the processor to leave the halted state.

Software should ensure that the kernel and interrupt stacks are mapped valid, writable (at least to kernel mode), and modified, so as to avoid kernel-stack-not-valid aborts and interrupt-stack-not-valid halts.

A machine-check exception indicates that the processor detected an internal error. As is usual for exceptions, machine-check is taken regardless of current IPL. The machine-check exception vector bits<1:0> must specify 1 or the operation of the processor is UNDEFINED. The exception is taken on the interrupt stack, and IPL is raised to 1F (hex).

The processor pushes a machine-check stack frame onto the interrupt stack, consisting of a count longword, an implementation-dependent number of error report longwords, and a PC and PSL. The count longword reports the number of bytes of error report pushed. For example, if 4 longwords of error report are pushed, the count longword will contain 16 (decimal). An example machine-check stack frame is shown in Figure 5-6.

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.

```
+---------------------------------------------------------------+
|                     sign extended code                        | :(SP)
+---------------------------------------------------------------+
|                   PC of next instruction                      |
+---------------------------------------------------------------+
|                          old PSL                              |
+---------------------------------------------------------------+
```

Figure 5-5 CHMx Stack Frame

```
+---------------------------------------------------------------+
|                      00000010 (hex)                           | :(SP)
+---------------------------------------------------------------+
|                1st longword of error report                   |
+---------------------------------------------------------------+
|                2nd longword of error report                   |
+---------------------------------------------------------------+
|                3rd longword of error report                   |
+---------------------------------------------------------------+
|                4th longword of error report                   |
+---------------------------------------------------------------+
|                             PC                                |
+---------------------------------------------------------------+
|                             PSL                               |
+---------------------------------------------------------------+
```

Figure 5-6 An Example Machine-Check Stack Frame

```
 31                                             9 8             0
+-----------------------------------------------+---------------+
|               page address of SCB             |      MBZ      |
+-----------------------------------------------+---------------+
```

Figure 5-7 System Control-Block Base

5.5 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 PSL<T> = 1 and PSL<TP> = 0, and 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, saving PC and PSL (with TP=1), loading a new PC from the overflow trap vector, and creating a new PSL.

3. The interrupt sequence is initiated, saving the PC and PSL appropriate to the overflow-trap service routine, loading a new PC from the interrupt 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 with REI.

5. The interrupt service routine runs and then exits with REI.

6. The overflow-trap service routine runs and then 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 PSL<TP>=0.

8. Trace service routine runs and then exits with REI.

9. The next instruction is executed.

This sequence is accomplished by the following operation between instructions:

```
        ! Here at completion of instruction, including
        !  at end of REI from an exception or interrupt routine.

   1$:  {possibly take interrupts or console halt};
        ! If so, PSL<TP> is not modified before PSL is saved.

        if PSL<TP> EQLU 1 then      ! If trace pending, then fault.
                begin               ! Trace fault takes precedence
                PSL<TP> <- 0;       !  over other exceptions.
                {initiate trace fault};
                end;

        {possibly take interrupts or console halt};
        ! If so, 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.
        ! PSL<FPD> is tested here, thus PSL<TP> takes
        !  precedence over PSL<FPD> if both are set.
        if {instruction faults} OR {an interrupt or console halt
            is taken before end of instruction} then
                begin
                {back up PC to start of opcode};
                {either set PSL<FPD> or back up all general
                 register side effects};
                PSL<TP> <- 0;
                {initiate exception or interrupt};
                end;

        if {arithmetic trap needed and no other abort or trap}
                then {initiate arithmetic trap};

        ! Note: All instructions end by flowing
        !  through 1$, thus the REI from a service
        !  routine will return to 1$.
```

5.6 SYSTEM CONTROL BLOCK

The system control block is a data structure in memory, containing the vectors by which exceptions and interrupts are dispatched to the appropriate service routines. Table 5-5 shows the interrupt and exception vectors in the SCB.

5.6.1 System Control-Block Base

The system control-block base (SCBB) is a privileged register containing the address of the system control block, which must be page-aligned. Figure 5-7 shows the SCBB. SCBB may contain either a virtual or a physical address, depending on the processor implementation.

For a processor with a 24-bit, 30-bit, 32-bit one-to-one, or 32-bit sign-extended physical address mode, the processor locates the SCB in physical memory and holds a physical address in SCBB. For a processor with a 34-bit physical address mode, the processor may locate the SCB in either physical or virtual memory and holds either a physical or virtual address in SCBB respectively.

The entire SCB must reside in the portion of physical memory that is accessible by virtual address translation when memory management is off. For example, on a processor which translates a virtual address to a 34-bit physical address by the preferred method described in 4.2.1, software must ensure that the SCB resides within the first 512 megabytes of physical memory.

Processor initialization leaves the contents of SCBB UNPREDICTABLE.

If the SCBB points to I/O space or nonexistent memory when an exception or interrupt occurs, the operation of the processor is UNDEFINED. If SCBB contains a virtual address, the page or pages containing the SCB must be valid and accessible to kernel mode, or the operation of the processor is UNDEFINED.

5.6.2 Interrupt and Exception Vectors

An interrupt or exception vector is a longword in the SCB. The processor examines the vector when an exception or interrupt occurs in order 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-dependent microcode there. If writable control store does not exist or is not loaded, the operation is UNDEFINED.

3 - Operation UNDEFINED. Reserved to DIGITAL.

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 of the processor is UNDEFINED. Emulation exceptions are serviced on the current stack. Bits <1:0> in the emulation exception vectors must be zero or the operation of the processor is UNDEFINED.

The assignment of SCB offsets and priority levels for controllers, adapters, and other devices connecting to the system bus is implementation dependent. Typically, interrupt priority levels 14 through 17 (hex) are used to signal I/O device, controller, and adapter events. Typically, one interrupt vector is assigned to each priority level for each adapter.

The use of second or third SCB pages (offsets 200 through 3FC and 400 through 5FC) is implementation dependent. In some processors (the VAX-11/750 and VAX-11/730, for example) UNIBUS devices interrupt the processor directly, and the second SCB page contains the UNIBUS device vectors. When a UNIBUS device connected to such a system requests an interrupt, the vector is determined by adding 200 (hex) to the vector supplied by the device. If a second UNIBUS adapter is installed, the third SCB page contains its device vectors, and 400 (hex) is added to the vector supplied by the device attached to the second UNIBUS interconnect. Only device vectors in the range 0 through 1FC (hex) are allowed. Interrupt priority levels 14 through 17 (hex) correspond to UNIBUS levels BR4 through BR7 if the system has a UNIBUS interconnect.

Table 5-5: System-Control-Block Vectors

Offset	Name	Type	Parameters	Notes
00	passive release	interrupt		IPL is that of the request.
04	machine check	exception	*	Number of parameters is implementation-dependent.
08	kernel stack not valid	abort	0	
0C	power fail	interrupt		IPL is 1E.
10	reserved or privileged instruction	fault	0	Opcodes reserved to DIGITAL and privileged instructions.
14	customer reserved instruction	fault	0	XFC instruction.
18	reserved operand	fault or abort	0	
1C	reserved addressing mode	fault	0	
20	access-control violation	fault	2	Virtual address and fault parameter are pushed.
24	translation not valid	fault	2	Virtual address and fault parameter are pushed.
28	trace pending	fault	0	
2C	breakpoint instruction	fault	0	
30	compatibility	fault or abort	1	A type code is pushed.
34	arithmetic	trap or fault	1	A type code is pushed.
38	VM emulation	trap	5	Opcode, PC, and up to three operands are pushed.
3C	modify	fault	2	Virtual address and fault parameter are pushed.
40	CHMK	trap	1	The operand word is sign-extended and pushed.
44	CHME	trap	1	The operand word is sign-extended and pushed.
48	CHMS	trap	1	The operand word is sign-extended and pushed.
4C	CHMU	trap	1	The operand word is sign-extended and pushed.
50 - 60	reserved for bus or memory error	interrupt		IPL is implementation dependent.
64	unused			Reserved to DIGITAL.
68	vector processor disabled	fault	0	See Chapter 14 for details
6C - 7C	unused			Reserved to DIGITAL.
80	interprocessor interrupt	interrupt		IPL is 16 (hex).
84	software level 1	interrupt		IPL is 1.
88	software level 2	interrupt		IPL is 2. Ordinarily used for AST delivery.
8C	software level 3	interrupt		IPL is 3. Ordinarily used for process scheduling.
90 - BC	software levels 4 - F	interrupt		Vector corresponds to IPL.
C0	interval timer	interrupt		IPL is 16 or 18 (hex), implementation-dependent.
C4	unused			Reserved to DIGITAL.
C8	instruction emulation	trap	10	FPD clear. Emulation frame is pushed.
CC	suspended emulation	fault	0	FPD set.
D0 - DC	unused			Reserved to DIGITAL.
E0 - EC	unused			Reserved to customers.
F0	console storage receive	interrupt		11/750 and 11/730. IPL is implementation-dependent.
F4	console storage transmit	interrupt		11/750 and 11/730. IPL is implementation-dependent.
F8	console terminal receive	interrupt		IPL is 14 (hex).
FC	console terminal transmit	interrupt		IPL is 14 (hex).
100-13C	adapter vectors	interrupt		IPL is 14 (hex). Implementation-dependent.
140-17C	adapter vectors	interrupt		IPL is 15 (hex). Implementation-dependent.
180-1BC	adapter vectors	interrupt		IPL is 16 (hex). Implementation-dependent.
1C0-1FC	adapter vectors	interrupt		IPL is 17 (hex). Implementation-dependent.
200-3FC	device vectors	interrupt		Implementation-dependent.
400-5FC	device vectors	interrupt		Implementation-dependent.

Table 5-6: Indication of Current Stack Pointer

Stack Pointer	Mnemonic	PSL<IS>	PSL<CUR_MOD>
Interrupt stack pointer	ISP	1	0
Kernel stack pointer	KSP	0	0
Executive stack pointer	ESP	0	1
Supervisor stack pointer	SSP	0	2
User stack pointer	USP	0	3

5.7 STACKS

At any time, the processor is either in a process context (and PSL<IS> = 0) in one of four modes (kernel, executive, supervisor, user), or is in the system-wide interrupt service context (and PSL<IS> = 1) that operates with kernel privileges. There is a stack pointer copy associated with each of these five states, and an internal processor register (IPR) to hold each stack pointer copy. Any time the processor changes from one of these states to another, SP (R14) is stored in the copy associated with the old state and SP is loaded from the copy associated with the new state. The five stack pointers are accessible as internal processor registers.

KSP	Kernel-mode stack pointer
ESP	Executive-mode stack pointer
SSP	Supervisor-mode stack pointer
USP	User-mode stack pointer
ISP	Interrupt stack pointer

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.

5.7.1 Stack Residency

The user, supervisor, and executive mode stacks do not need to be resident. Kernel-mode code can bring in or allocate process stack pages as translation-not-valid faults occur. The kernel stack for the current process and the interrupt stack (which is process-independent), however, must be resident, accessible, and modified. Translation-not-valid, access-control-violation, and modify faults occurring on references to either of these stacks are regarded as serious system failures. (Because the console may use 8 bytes of interrupt-stack space when starting a processor, the 8 bytes above the top of the interrupt stack, -8(SP) through -1(SP), must also be resident.) If any of these faults occur while pushing exception or interrupt state onto the interrupt stack, the processor halts. If translation-not-valid or access-control-violation fault occur while pushing exception or interrupt state onto the kernel stack, the processor aborts the current sequence and initiates kernel-stack-not-valid abort on hardware level 1F (hex). If modify fault occurs while pushing exception or interrupt state onto the kernel stack, the processor is allowed to do one of two things: set the PTE<M> bit for that page and continue normal operation; or generate a kernel-stack-not-valid abort on hardware level 1F (hex). The processor may implement the first option even if it supports modify fault.

The behavior described above does not mean that every possible reference to the kernel and interrupt stacks is checked, but rather that the processor will not loop on these faults.

It is not necessary that the kernel stack for a process other than the current one be resident, but it must be resident before that process is selected to run by the software's process dispatcher. Further, any mechanism that uses memory-management faults to gather process statistics, for instance, must exercise care not to fault kernel-stack or interrupt-stack pages.

5.7.2 Stack Alignment

Except on CALLS and CALLG instructions, the hardware makes no attempt to align the stacks. For best performance on all processors, the software should align the stack on a longword boundary and allocate the stack in longword increments. The convert-byte-to-long (CVTBL and MOVZBL), convert-word-to-long (CVTWL and MOVZWL), convert-long-to-byte (CVTLB), and convert-long-to-word (CVTLW) instructions are recommended for pushing bytes and words on the stack and popping them off in order to keep it longword aligned.

5.7.3 Stack Status Bits

The interrupt stack bit and current-mode bits in the PSL specify which of the five stack pointers is currently in use, as shown in Table 5-6.

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. If the current PSL<IS> is 1 or if the low bits of the vector are 01 (binary), then the interrupt stack is used. If the current PSL<IS> is 0 and the low bits of the vector are 00, then the kernel stack is used. Values 10 (binary) and 11 (binary) of the vector<1:0> are used for other purposes. Refer to the section "System Control Block" earlier in this chapter for details.

5.7.4 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 -- depending on the values of the current mode and IS bits in the PSL. Some processors may implement these five stack pointers as five internal processor registers. Other processors may store the four per-process stack pointers in memory in the PCB and store only the interrupt stack pointer in an internal register (see Chapter 6). In either case, software can access any of the five stack pointers with 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 internal processor register space by an MTPR or MFPR instruction.

\When the processor is running, this applies only to the KSP and ISP, since these instructions require kernel privilege. This means that MTPR and MFPR must access the SP (general register 14) if the KSP (if IS=0) or the ISP (if IS=1) is being referenced. When the processor is halted, the console may try to read any of the stack pointers regardless of the current mode.\

If the four process stack pointers are implemented as registers, then these instructions are the only method for accessing the stack pointers of the current process. See Chapter 8 for conventions to be followed when referencing other per-process registers in the internal processor register space.

The internal processor register numbers were chosen to be the same as PSL<26:24>. The previous stack pointer is the same as PSL<23:22> unless PSL<IS> is set. If PSL<IS> is set, the previous mode cannot be determined from the PSL since interrupts always clear PSL<23:22>. Processor initialization leaves the contents of all stack pointers UNPREDICTABLE.

5.7.5 Memory Above Top-of-Stack

To ensure correct operation, software must assume that preceding the initiation of every instruction a processor will set the contents of all writable bytes between -1(SP) and -512(SP) to UNPREDICTABLE values. Bytes that cannot be written because of access-control violation, translation-not-valid, or modify fault are unaffected. So, for example:

```
        MOVW    register_mask, -(SP)
        PUSHR   (SP)+                   ; Produces predictable results

        MOVW    register_mask, -2(SP)
        PUSHR   -2(SP)                  ; Produces UNPREDICTABLE results

        MOVAB   arg_list, 10(SP)
        CALLG   @10(SP), dst            ; Produces predictable results

        MOVAB   arg_list, -10(SP)
        CALLG   @-10(SP), dst           ; Produces UNPREDICTABLE results

        CALLG   (SP)+, @(SP)+           ; Produces predictable results
```

Processors may take advantage of this in interpreting instructions that push operands onto the stack, by beginning to write the operands above the top of the stack without having first probed to ensure that all operands can be written. Processors are not generally allowed to write memory before ensuring that the instruction will complete, but writes to operands above the top of the stack are explicitly excluded from this rule.

5.8 INITIATE EXCEPTION OR INTERRUPT

```
        <none>          Initiate Exception or Interrupt
```

Operation:

```
        ! Read the vector into a temporary register, and check it
        ! for validity.  The vector number is determined by the
        ! exception or interrupt type.
        {disable interrupts};
        vector <- SCB[vector_number];
        case vector<1:0> of
                0:      if {machine check OR kernel-stack-not-valid}
                                then {UNDEFINED};
                1:      if {CHMx OR instruction emulation exception}
                                then {UNDEFINED};
                2:      if {user microcode exists and is loaded}
                                then {enter user microcode}
                                else {UNDEFINED};
                3:      {UNDEFINED};
        end;

        ! Save the current PSL in a temporary register.
        saved_PSL <- PSL;

        ! Create and load a new PSL.
        case {exception or interrupt type} of
                {interrupt}:
                        begin
                        PSL<CM,TP,FPD,DV,FU,IV,T,N,Z,V,C> <- 0;
                        PSL<VM> <- 0;
                        PSL<CUR_MOD,PRV_MOD> <- 0;
                        if vector <1:0> EQLU 1
                                then PSL<IS> <- 1
                                else PSL<IS> <- saved_PSL<IS>;
                        PSL<IPL> <- new_IPL;
                        end;
                ! Note that CHMx exceptions when PSL<VM> is set
                ! are turned into VM-emulation traps (see CHMx
                ! instruction.)
                {CHMx}:
                        begin
                        PSL<CM,TP,FPD,DV,FU,IV,T,N,Z,V,C> <- 0;
                        PSL<VM> <- 0;
                        PSL<CUR_MOD> <- new_mode;
                        PSL<PRV_MOD> <- saved_PSL<CUR_MOD>;
                        PSL<IS> <- saved_PSL<IS>;
                        PSL<IPL> <- saved_PSL<IPL>;
                        end;
        {instruction emulation exception}:
                begin
                ! If VM bit set, change instruction
                ! emulations into VM-emulation exceptions
                ! after pushing operands onto current
                ! stack.
                if {PSL<VM> EQLU 1} then
                        {initiate VM-emulation trap};
                PSL<CM,TP,FPD,DV,FU,IV,T> <- 0;
                PSL<VM> <- 0;
                PSL<CUR_MOD> <- saved_PSL<CUR_MOD>;
                PSL<PRV_MOD> <- saved_PSL<PRV_MOD>;
                PSL<IS> <- saved_PSL<IS>;
                PSL<IPL> <- saved_PSL<IPL>;
                PSL<N,Z,V,C> <- saved_PSL<N,Z,V,C>;
                end;
        otherwise                 ! (Other exceptions.)
                begin
                ! VM-emulation exception falls into this
                ! category.
                PSL<CM,TP,FPD,DV,FU,IV,T,N,Z,V,C> <- 0;
                PSL<VM> <- 0;
                PSL<CUR_MOD> <- 0;
                PSL<PRV_MOD> <- saved_PSL<CUR_MOD>;
                if vector<1:0> EQLU 1
                        then PSL<IS> <- 1
                        else PSL<IS> <- saved_PSL<IS>;
                if vector<1:0> EQLU 1
                        then PSL<IPL> <- 31
                        else PSL<IPL> <- saved_PSL<IPL>;
                end;
        end;

! If necessary, save the current stack pointer and load a
! new one.
if saved_PSL<IS> EQLU 0 then
        begin
        IPR[saved_PSL<CUR_MOD>] <- SP;
        SP <- IPR[ PSL<IS>'PSL<CUR_MOD> ];
        end;

! Push PC, the saved PSL, and any parameters onto the new
! stack, in the new mode.  If a memory management fault
! occurs while pushing onto the new stack, initiate an
! abort or halt.  (This will not occur with CHMx or
! instruction emulation exceptions, since they explicitly
! probe the destination stack and fault if it is invalid
! or inaccessible.)
-(SP) <- saved_PSL;
-(SP) <- PC;
{push parameters if any};
```

```
        ! Load PC with the address of the exception or interrupt
        ! handler.
        PC <- vector<31:2> ' 0<1:0>;
        {enable interrupts};
        ! Software interrupt clear the software-interrupt-pending
        ! bit.
        if {software interrupt} then SISR< PSL<IPL> > <- 0;
```

Condition Codes:

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

Exceptions:

kernel-stack not valid
interrupt-stack not valid

Description:

The vector associated with the exception or interrupt is read from the system control block. The current PSL is saved and a new PSL is created and loaded. If this is an interrupt, the new PSL has all fields cleared except IS and IPL. IPL is raised to the priority level of the interrupt request. IS is set to 1 if the low bits of the vector contain 01 (binary); otherwise, it is unchanged from the old PSL. If this is a CHMx exception, CUR_MOD is loaded with the new mode, PRV_MOD is loaded with the old value of CUR_MOD, IS and IPL are retained from the old PSL, and all other fields are cleared. If this is an emulation exception and if the processor is executing a VM, transform the instruction emulation exception into a VM-emulation exception. Otherwise, if this is an emulation exception, CUR_MOD, PRV_MOD, IS, IPL, and the condition codes are all retained from the old PSL, and all other fields are cleared. If this is any other kind of exception, PRV_MOD is loaded with the old value of CUR_MOD, IS and IPL are loaded according to the low bits of the vector, and all other fields are cleared. If the low bits of the vector are 01 (binary) then IS is loaded with 1 and IPL is raised to 31; otherwise, IS and IPL are retained from the old PSL. Unless the processor is already running on the interrupt stack, the old stack pointer is saved and a new one is loaded. The saved PSL and the PC are pushed onto the stack, along with any exception parameters. PC is loaded with the address of the interrupt or exception service routine indicated by bits <31:2> of the vector. Table 5-7 summarizes the state transitions CHMx, REI, interrupts, and exceptions cause.

Notes:

1. Interrupts are disabled during this sequence.

2. 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.

3. After an abort, all the explicit and implicit operands of the aborted instruction are UNPREDICTABLE. The PC pushed on the stack points to the opcode of the aborted instruction, unless the instruction modified PC in a way that produces UNPREDICTABLE results.

4. After an abort or fault or interrupt that pushes a PSL with FPD set, 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. This implies that processes stopped with FPD set cannot be resumed on processors of a different type or engineering change level.

5. If the processor gets an access-control-violation or translation-not-valid condition while attempting to push information on the kernel stack, a kernel-stack-not-valid abort is initiated instead, and IPL is raised to 31. The PSL and PC saved on the interrupt stack are those that would have been pushed on the kernel stack by the original exception. Additional information, if any, associated with the original exception is lost. If vector<1:0> for kernel-stack-not-valid abort is 0, the operation of the processor is UNDEFINED. (Kernel stack not valid will not occur with CHMx or instruction emulation exceptions, since they explicitly probe the destination stack and fault if it is invalid or inaccessible.)

6. If the processor gets an access-control-violation or 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 ensured to be correct for subsequent analysis. The PSL and PC have the values that would have been pushed on the interrupt stack.

7. The value of PSL<TP> that is saved on the stack is as follows:

fault	clear
trace	clear
interrupt	clear (if FPD set) from PSL<TP> (if after traps, before trace)
abort	UNPREDICTABLE
trap	from PSL<TP>
CHMx	from PSL<TP>
BPT, XFC	clear
reserved or privileged instruction	clear

8. The value of PC that is saved on the stack points to the following:

fault	instruction faulting
trace	next instruction to execute (instruction at the beginning of which the trace fault was taken)
interrupt	instruction interrupted or next instruction to execute
abort	instruction aborting or detecting kernel-stack-not-valid (not ensured on machine check)
trap	next instruction to execute
CHMx	next instruction to execute
BPT, XFC	BPT, XFC instruction
reserved or privileged instruction	privileged instruction

   Table 5-7 summarizes the state transitions caused by interrupts, exceptions, and REI.

9. Microcode is inside the system's security boundary. This means that any user who can change the contents of microcode can penetrate the system. In secure applications, it is a requirement for system security that customers not have access to microcode, and bits <1:0> of an SCB vector should never contain a 2.

10. If the processor is in VM mode, both instruction-emulation traps and suspended-emulation faults are handled specially. In both types of exceptions, an exception frame is pushed onto the current stack, and a VM-emulation trap is taken in kernel mode. For more details, see Chapter 12, Virtual Machines.

Table 5-7: Processor State Transitions

	Final State					
Initial State	User Mode	Super Mode	Exec Mode	Kernel Stack, IPL = 0	Kernel Stack, IPL > 0	Interrupt Stack
User Mode	REI or CHMU	CHMS	CHME	CHMK or e(0)	i(0)	e(1) or i(1)
Super Mode	REI	REI, CHMU, or CHMS	CHME	CHMK or e(0)	i(0)	e(1) or i(1)
Exec Mode	REI	REI	REI, CHMU, CHMS, or CHME	CHMK or e(0)	i(0)	e(1) or i(1)
Kernel Stack, IPL = 0	REI	REI	REI	REI, CHMx, LDPCTX, e(0), or MTPR IPL	MTPR IPL or i(0)	e(1), i(1), or SVPCTX
Kernel Stack, IPL > 0	REI	REI	REI	REI or MTPR IPL	REI, CHMx, LDPCTX, e(0), or i(0), or MTPR IPL	e(1), i(1), or SVPCTX
Interrupt Stack	REI	REI	REI	REI	REI or LDPCTX	REI, SVPCTX, MTPR IPL, exception or interrupt

e(0) means exception with vector<1:0> = 0
e(1) means exception with vector<1:0> = 1
i(0) means interrupt with vector<1:0> = 0
i(1) means interrupt with vector<1:0> = 1

5.9 INSTRUCTIONS RELATED TO EXCEPTIONS AND INTERRUPTS

REI   Return from Exception or Interrupt

Format:

```
Opcode  {(SP)+.r*}
```

Operation:

```
tmp1 <- (SP)+;  ! Pick up saved PC
tmp2 <- (SP)+;  ! and PSL

! Check that tmp2 is a valid PSL
if {tmp2<IS> EQLU 1 AND tmp2<IPL> EQLU 0} OR
   {tmp2<IPL> GTRU 0 AND tmp2<CUR_MOD> NEQU 0} OR
   {tmp2<PRV_MOD> LSSU tmp2<CUR_MOD>} OR
   {tmp2<PSL_MBZ> NEQU 0} then {reserved operand fault};

if {compatibility mode implemented} then
        begin
        if {tmp2<CM> EQLU 1} AND
           {{tmp2<FPD,IS,DV,FU,IV> NEQU 0} OR
            {tmp2<CUR_MOD> NEQU 3}} then
                {reserved operand fault};
        end
else if {tmp2<CM> EQLU 1} then {reserved operand fault};

! Case 1:  REI executed within a virtual machine
if {PSL<VM> EQLU 1} then
    begin
    ! Check tmp2 against VMPSL
    if {tmp2<CUR_MOD> LSSU VMPSL<CUR_MOD>} OR
       {tmp2<IS> EQLU 1 AND VMPSL<IS> EQLU 0} OR
       {tmp2<IPL> GTRU VMPSL<IPL>} then
                {reserved operand fault};
    {VM-emulation trap};
    ! Exception frame contains tmp1 and tmp2 as
    ! operand 1 and operand 2, respectively, and the
    ! PC and PSL at the time of executing the REI as
    ! PC and PSL, respectively.
    ! Note that REI in a virtual machine to a PSL with
    ! <VM> bit set is part of this case, and software
    ! should specifically test for it.
    end

! Case 2:  REI executed in a real machine
if {tmp2<CUR_MOD> LSSU PSL<CUR_MOD>} OR
   {tmp2<IS> EQLU 1 AND PSL<IS> EQLU 0} OR
   {tmp2<IPL> GTRU PSL<IPL>} OR
   {tmp2<VM> EQLU 1 AND PSL<CUR_MOD> NEQU 0} then
                {reserved operand fault};

if PSL<IS> EQLU 1 then ISP <- SP  !save old stack pointer
                  else IPR[ PSL<CUR_MOD> ] <- SP;
        if PSL<TP> EQLU 1 then tmp2<TP> <- 1;   !TP <- TP or
                                                ! stack TP
        PC <- tmp1;
        PSL <- tmp2;
        if PSL<IS> EQLU 0 then
                begin
                SP <- IPR[ PSL<CUR_MOD> ];      !switch stack
                if PSL<CUR_MOD> GEQU ASTLVL     !check for AST
                                                ! delivery
                        then {request interrupt at IPL 2};
                end;
        {check for software interrupts};
        {clear instruction look-ahead};
```

Condition Codes:

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

Exception:

reserved operand
VM emulation

Opcode:

02	REI	Return from Exception or Interrupt

Description:

A longword is popped from the current stack and held in a temporary PC. A second longword is popped from the current stack and held in a temporary PSL. The popped PSL is checked for internal consistency. If the processor is running a VM, the popped PSL is compared with VMPSL to make sure that the transition from current VMPSL to popped PSL is allowed; then a VM-emulation trap is taken. If the processor is running a real machine, the popped PSL is compared with the current PSL to make sure that the transition from current PSL to popped PSL is allowed. If the processor is not in kernel mode and is attempting to return to a PSL with the VM bit set, a reserved operand fault occurs. 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 "Stack Status Bits" earlier in this chapter). The level of the highest-privileged AST is checked against the current mode to see whether a pending AST can be delivered (see Chapter 6). Execution resumes with the instruction being executed at the time of the exception or interrupt.

After completing an REI, a processor will correctly execute a modified instruction stream (see section 7.2.2).

Notes:

1. \ As a result of executing an REI, a processor must ensure that its prefetched or buffered copies of the instruction stream correspond with the modified instruction stream as seen by that processor. \

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

3. As usual for faults, if translation not valid or access-control violation occurs while popping PC or PSL from the stack, the stack pointer is restored as part of the initiation of the fault.

4. REI to compatibility mode results in a reserved operand fault if compatibility mode is not implemented.

5. \If the popped PSL specifies interrupt stack and specifies a mode other than kernel, REI will fault even though there is no explicit check for this condition. Imagine that the condition holds; that is, {tmp2<IS> EQLU 1 AND tmp2<CUR_MOD> NEQU 0}. If tmp2<IPL> EQLU 0, then the test {tmp2<IS> EQLU 1 AND tmp2<IPL> EQLU 0} will cause the reserved operand fault. Otherwise, the test {tmp2<IPL> GTRU 0 AND tmp2<CUR_MOD> NEQU 0} will cause the reserved operand fault.\

6. The processor enters a virtual machine by executing an REI instruction. When the REI is executed, the value of PSL is constrained by the value of VMPSL. The constraints are:

   ```
   PSL<CUR_MOD> EQLU MAXU( VMPSL<CUR_MOD>, 1 );
   PSL<PRV_MOD> EQLU MAXU( VMPSL<PRV_MOD>, 1 );
   PSL<IPL> EQLU 0;  and
   PSL<IS> EQLU 0.
   ```

   If any of these constraints does not hold, operation of the processor is UNDEFINED.

7. When the processor is executing a virtual machine and the VM executes an REI instruction and the VM bit in the PSL on the stack is clear (case 1 above), REI compares tmp2 and VMPSL to ensure that the transition from VMPSL to tmp2 is allowed. This is done so that software implementing virtual machines need not make those checks.

8. If virtualization is not implemented by some processor, and an attempt is made to REI to a PSL whose VM bit is equal to 1, a reserved-operand exception is taken.

```
        CHM     Change Mode

Purpose:        request services of more privileged software

Format:

        opcode  code.rw {-(ySP).w*}

                where y=MINU(x, PSL<CUR_MOD>)
                  where x=Mode selected by opcode (K=0, E=1, S=2, U=3)

Operation:

        if PSL<VM> EQLU 1 then
                {initiate VM-emulation trap};   !exception frame
                                                !contains code
        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

        IPR[ PSL<CUR_MOD> ] <- SP;              !save old stack pointer
        tmp4 <- IPR[ tmp2 ];                    !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-control-violation fault};
        if {translation not valid} then
            {initiate translation-not-valid fault};
        if {modify} then
            {initiate modify 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;

Exception:

        halt
        VM emulation

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 (decreases the access mode) or leaves it unchanged.

If the processor is executing a virtual machine, a VM-emulation trap occurs. Otherwise a change in access mode occurs.

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. Figure 5-5 illustrates the new stack's appearance after execution.

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 is NEQU 0, then the operation is UNDEFINED.

Notes:

1. As usual for faults, any memory-management fault saves PC and PSL and leaves SP as it was at the beginning of the instruction (except for any pushes onto the kernel stack). Note, though, that addresses just off the top of the target stack may have been written (see section 5.7.5).

2. By software convention, negative codes are reserved to DIGITAL and DIGITAL's 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.
```

Change History:

Revision J. Rich Brunner and Tim Leonard, December 1989.

- o If modify fault occurs while pushing exception or interrupt state onto the kernel stack, the processor is allowed to do one of two things: set the PTE<M> bit for that page and continue normal operation; or generate a kernel-stack-not-valid abort on hardware level 1F (hex). The processor may implement the first option even if it supports modify fault.
- o Add vector processor disabled fault to SCB (68 hex).
- o ECO 101 -- Allow writes above top-of-stack.
- o REI forces processor to execute modified instruction stream.
- o Location and type of address in SCBB determined by physical address mode.
- o The entire SCB must reside in the portion of physical memory that is accessible by virtual address translation when memory management is off.

Revision H. Tim Leonard, May 1987.

- o Support virtual machines.
- o Make modify fault flush the faulting PTE from the TB.
- o Note that floating overflow trap is obsolete.
- o Add extended physical addresses, modify fault, and allow base registers to contain virtual addresses.

Revision F. Al Thomas, November 1986.

- o Replace subset-emulation trap with instruction-emulation trap.
- o Add that the console may use memory bytes -8(ISP)..-1(ISP) in the process of starting the processor.

Revision E. Al Thomas, September 1986.

- o Add sections from the Memory and I/O chapter that deal with interrupts, errors, and restartability.
- o List vector 80 as interprocessor interrupt.
- o Clarify probe of stack during instruction-emulation exceptions.
- o Add implied operands for REI and CHMx.

Revision D1. Tim Leonard, March 1986.

- o Include CHMx trap in the list of exceptions.
- o Add the new description of Initiate Exception or Interrupt. It should have been included in Revision D.

Revision D. Tim Leonard, March 1985.

- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Add "passive release" interrupt to SCB vectors.
- o Change the name of "Opcode Reserved To Digital Fault" to "Reserved or Privileged Instruction Fault", to match the rest of the manual.
- o Rename the emulation exceptions.
- o Display the table explaining tracing in a different format.

- o MTPR #0, IPL while on the interrupt stack is UNDEFINED.
- o Kernel-stack-not-valid abort must push the PC and PSL of the original exception.
- o Remove redundant check in REI: {tmp2<IS> EQLU 1 AND tmp2<CUR_MOD> NEQU 0}.
- o REI to compatibility mode results in a reserved operand fault if compatibility mode is not implemented.
- o Mechanism for enabling and disabling XFC fault is implementation dependent.
- o Timer IPL can be 16 or 18 (hex).
- o Move PSL description to Chapter 2.
- o Change description of vector CC (emulation exception with FPD set) to say no operands are pushed, so as to match Appendix E.

Revision 8. Dileep Bhandarkar, 26 July 1982.

- o POLY reserved operand exception should be a fault.
- o Add clearing of instruction look-ahead requirement to REI operation description.
- o Kernel-stack-not-valid is UNDEFINED when exception vector <1:0> is 0.
- o Define what is UNPREDICTABLE after an abort.

Revision 7. Dileep Bhandarkar, 12 May 1980.

- o Update SCB layout.
- o Clarify trace fault.
- o Extend SCB for 11/750 UNIBUS interrupts.

Revision 6, add floating faults. Dileep Bhandarkar, 31 January 1979.

- o Floating Fault Enable.
- o REMQUE and INSQUE are unaligned.
- o ECO IS, IPL for exceptions.

Revision 5. Peter Conklin and Ted'Taylor, 30 July 1978.

- o Add software mnemonics for arithmetic traps.
- o REMQUE and INSQUE are aligned (ECO).
- o Clean up Initiate Exception or Interrupt.

Revision 4. Peter Conklin, 20 April 1977.

- o Add reserved length on EDIT (EDITPC ECO).
- o Make kernel-stack-not-valid IPL be 1F, powerfail IPL be 1E.
- o Correct numerous typos including residual -16 and -14.
- o Add POLYx reserved operand.
- o Change "disaster" to "urgent".
- o Compatibility mode is a fault except odd address which is an abort. Reconcile codes with chapter 10.
- o Trace is a trap (TP is a fault).
- o IPL 1F is for exception vector<1:0>=1..
- o IPL 1E is for powerfail only.
- o If interrupt routine modifies FPD, registers, or CC, then results are UNPREDICTABLE.
- o Add console terminal and interval timer vectors and IPL.
- o Change "STAR device" to "NEXUS".
- o CHMK and REI do not get KS not valid or IS not valid.

- o REI verifies IPL NEQU 0 if IS NEQU 0.
- o Add BICPSW and BISPSW to reserved operand list.
- o Remove terms ISR, ESR, WCS.
- o Add flow for initiation of exception or interrupt.
- o Add between instruction flow.
- o If FPD=1 saving PSL, then TP cleared. Verify on REI.
- o Verify on REI that if IS=1, then current mode=0.
- o Correct PROBEW in CHMx.
- o Reserve negative CHMx codes to CSS, customers. Change table to document that transitions are legal in all modes to IS=0 for interrupts and IS=1 for exceptions.
- o SCBB bit 30 is MBZ also.
- o Clarify the PCB stack pointers are not maintained.
- o Add invalid digit to reserved operand list.
- o Don't allow IPL>0 if current mode NEQ 0.
- o Document FC and FFFF as permanently reserved.
- o Document saved PC on all exceptions and interrupts.
- o Document that access-control-violation fault takes precedence over translation-not-valid fault.
- o Add interrupts example.
- o Machine-check is on a best effort basis.
- o Move IPL here from chapter 9. IPL<31:5> is ignored on write, returned 0.
- o SIRR ignores <31:5>; ignores <4:0> = 0; reserved aborts if <4:0> GTRU 15.
- o Add table of arithmetic trap types.
- o Change reserved operand reference from CVTNP to CVTTP, CVTSP (ECO).
- o Clarify FP and SP unpredictability on aborts.
- o Merge divide by zero trap; add index trap (INDEX ECO).
- o Note that faults do not restore everything, only enough.
- o Another interrupt versus exception distinction is the previous mode field.
- o Condition codes may change on reserved operand.
- o STAR violates reset not clear halt reason.
- o If a routine sets FPD, UNPREDICTABLE.
- o SCBB is really processor dependent.
- o Stack switch on exception or interrupt is optional if to same stack.
- o MTPR IPL changes states in state table.
- o Don't execute first instruction of interrupt routine if a higher priority interrupt is pending.
- o Check console halt with all interrupts.
- o Add that STAR memory errors on 1B.
- o Remove that interrupts cannot push on stack. The future might find it useful.
- o Interrupts and exceptions can be to shared process space!
- o Add that STAR traps PC, (PC), -(PC).
- o MTPR, MFPR now get stack pointers.
- o Add bootstrap initial settings of these registers.
- o POLY underflow occurs at end.
- o Add processor register reserved operands.
- o CALL clears saved T.
- o Add documentation of xSP registers.

- o Legal to REI with FPD and TP both set. TP takes precedence.
- o SIRR<31:4> are ignored.
- o On STAR, vector<1:0> = 2 with no WCS is a HALT.
- o On STAR, CHMx vector<1:0> is ignored.
- o Check ASTLVL in REI only if returning to IS=0.
- o Add explanation and rationale for T and TP.
- o Add usage note that IPL splits with all ISP above all KSP.
- o Add usage note for interrupt service routines to not drop IPL below original.
- o Add usage notes for debuggers.

Revision 3, approval by STAR task force. Jud Leonard, 3 June 1976.

- o Renumber interrupt levels, in hex, for hardware 10 to 1F and software 01 to 0F.
- o Provide names of Software Interrupt Request Register (SIRR) and Software Interrupt Summary Register (SISR).
- o Change "Numeric" to "Decimal".
- o Clarify fact that previous mode is cleared on interrupt, and loaded from current mode on exceptions.
- o Add description of software interrupt mechanism.
- o Add description of System Control Block (SCB), its base register (SCBB), and format of vectors.
- o Remove requirement to halt on exception if IS=1.
- o Combine cache parity error with machine-check. Fix machine-check to note that exception may be fault.
- o Change name of trace exception back to trap. Even though implemented as a fault, people think of it as a trap.
- o Fill out descriptions of arithmetic traps, giving type codes, result, and condition codes.
- o Eliminate decimal strings from reserved operand list, and illegal combinations in CALLx, RET; illegal PCB in LDPCTX; illegal register in MTPR, MFPR.
- o Combine memory error abort with machine-check.
- o Remove concept of programmable device vectors.

Revision 2, results of the April Task Force review. Jud Leonard, 7 May 1976.

- o Define IPLs -16 to 15, and clarify definitions.
- o Change all references to "exception stack" to "kernel stack".
- o Add descriptions of processor modes and stacks.
- o Eliminate trap-pending bits other than TP.
- o Redefine PSL.
- o Delete references to ISL.
- o Rewrite exception, interrupt and REI flow descriptions to work with new scheme.
- o Simplify T bit description for CHM scheme.
- o Eliminate address break and reserved address traps.
- o Rewrite REI description.
- o Add CHM instructions.
- o Add chart describing state transitions.
- o Fix introduction to eliminate misconception that exceptions are always synchronous and interrupts aren't. New version could still use word.
- o Remove concept of interrupt enables.

- Clarify stack residency and validity.
- Rename T bit to trace. Clarify how T and TP work for trace fault.
- Define effect of bits 1:0 of vector to select kernel stack, interrupt stack, or WCS.
- Redefine SCB vectors, eliminating and combining conditions, and add interrupt vectors.
- Change most aborts to faults or traps.
- Specify results from arithmetic traps; clarify floating overflow and underflow.
- Change Master Control Block (MCB) to System Control Block (SCB).
- Include checks of compatibility mode conditions in PSL of REI.
- Add example showing serialization of trap, interrupt, and trace.

Revision 1, initial distribution. Tom Hastings, October 1975.

CHAPTER 6

PROCESS STRUCTURE

A process is a single thread of execution. It is the basic scheduling 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, the program counter, the four per-process stack pointers, the base and length registers P0BR, P0LR, P1BR, and P1LR used to access the process virtual memory provided to the process, the address space number (ASN) associated with that process virtual memory, 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. 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.

6.1 PROCESS CONTEXT

Shown in Figure 6-1 is the process control block for the currently executing process. The PCB is pointed to by the process control block base (PCBB) register, an internal privileged register. Figure 6-2 shows the PCBB. When the processor is initialized, the contents of PCBB are UNPREDICTABLE.

For a processor with a 24-bit, 30-bit, 32-bit one-to-one, or 32-bit sign-extended physical address mode, the processor locates the PCB in physical memory and holds a physical address in PCBB. For a processor with a 34-bit physical address mode, the processor locates the PCB in virtual memory and holds a virtual address in PCBB. If the processor does hold a virtual address in PCBB, the PCB must not cross a page boundary, and the page containing the PCB must be valid, and the page must allow kernel write access, and (on processors that implement modify fault) the page must be marked "modified", or the operation of the processor is UNDEFINED.

Because the processor may use PCBB<n:7> (where n is the most significant bit of PCBB used by the processor) to uniquely identify a process, software must ensure that there is not more than one PCB in any aligned 128-byte block of memory (physical or virtual memory, depending on whether PCBB contains a physical or virtual address).

```
 31                                                            0
+---------------------------------------------------------------+
|                              KSP                              | :PCB
+---------------------------------------------------------------+
|                              ESP                              | +4
+---------------------------------------------------------------+
|                              SSP                              | +8
+---------------------------------------------------------------+
|                              USP                              | +12
+---------------------------------------------------------------+
|                              R0                               | +16
+---------------------------------------------------------------+
|                              R1                               | +20
+---------------------------------------------------------------+
|                               .                               |
|                               .                               |
|                               .                               |
+---------------------------------------------------------------+
|                              R9                               | +52
+---------------------------------------------------------------+
|                              R10                              | +56
+---------------------------------------------------------------+
|                              R11                              | +60
+---------------------------------------------------------------+
|                           AP (R12)                            | +64
+---------------------------------------------------------------+
|                           FP (R13)                            | +68
+---------------------------------------------------------------+
|                              PC                               | +72
+---------------------------------------------------------------+
|                              PSL                              | +76
+---------------------------------------------------------------+
|                             P0BR                              | +80
+---------+-----+---+-------------------------------------------+
|   MBZ   | AST |MBZ|                P0LR                       | +84
+---------+-----+---+-------------------------------------------+
|                               P1BR                            | +88
+-+-----------------+-------------------------------------------+
| |       MBZ       |                  P1LR                     | +92
+-+-----------------+---------------------------+---------------+
|                       ASN                     |    PRVCPU     | +96
+-----------------------------------------------+---------------+
```

Figure 6-1 Process Control Block (PCB)

Table 6-1: Contents of the Process Control Block

Name	Mnemonic	Offset (hex)	Extent
kernel stack pointer	KSP	0	<31:0>
executive stack pointer	ESP	4	<31:0>
supervisor stack pointer	SSP	8	<31:0>
user stack pointer	USP	C	<31:0>
general registers	R0 -- R13	10 -- 44	<31:0>
program counter	PC	48	<31:0>
processor status longword	PSL	4C	<31:0>
P0 base register	P0BR	50	<31:0>
P0 length register	P0LR	54	<21:0>
AST level	ASTLVL	54	<26:24>
P1 base register	P1BR	58	<31:0>
P1 length register	P1LR	5C	<21:0>
performance monitor enable	PME	5C	<31>
previous CPU	PRVCPU	60	<7:0>
address space number	ASN	60	<31:8>

```
 3
 1                                                         2 1 0
+-----------------------------------------------------------+-+-+
|                     address of the PCB                    |0|0|
+-----------------------------------------------------------+-+-+
```

Figure 6-2 Process Control-Block Base Register (PCBB)

```
 31                                                          1 0
+-------------------------------------------------------------+-+
|                                                             |P|
|                            MBZ                              |M|
|                                                             |E|
+-------------------------------------------------------------+-+
```

Figure 6-3 Performance-Monitor-Enable Register (PME)

```
 31                                                     3 2    0
+-------------------------------------------------------+------+
|                     MBZ; returns 0                    |ASTLVL|
+-------------------------------------------------------+------+
```

Figure 6-4 AST-Level Register (ASTLVL)

The 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. The PCB's contents are described in Table 6-1. To alter its P0BR, P1BR, P0LR, P1LR, ASN, ASTLVL or PME, a process must be executing in kernel mode. The process 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.

The ASTLVL and PME fields of the PCB may be contained in internal processor registers when the process is running.

The PRVCPU field is optional and need only be implemented in a processor that implements the ASN register. On such a processor, PRVCPU is used in the following way:

1. SVPCTX writes the contents of CPUID into PRVCPU.

2. LDPCTX compares CPUID with the value of PRVCPU and flushes the process-space TB state associated with the new value of the ASN register if they don't match.

3. Software can force the process-space TB state associated with a new value of ASN to be flushed, by writing 255 to PRVCPU and then executing LDPCTX. The PRVCPU value 255 is reserved for this purpose.

6.1.1 Performance Monitor Enable Register

The performance-monitor-enable (PME) register controls a signal visible to an external hardware performance monitor. PME allows the system to identify those processes for which monitoring is desired and so permits their behavior to be observed without interference caused by the activity of other processes. Figure 6-3 shows PME. Processor initialization sets PME to zero.

6.2 ASYNCHRONOUS SYSTEM TRAPS

Asynchronous system traps (AST) are a technique for notifying a process of events that are not synchronized with its execution and for initiating processing of asynchronous events with the least possible delay. This delay in delivery of the AST may be due to either process non-residence or to an access mode mismatch. The efficient handling of ASTs in the VAX system requires some hardware assistance to detect changes in access mode (PSL<CUR_MOD>). A process in any of the four access modes (kernel, executive, supervisor, and user) may receive

ASTs; 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, shown in Figure 6-4. ASTLVL contains 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 posted to cause delivery of the pending AST.

The software flow for AST processing follows:

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 again computed and moved to the PCB and ASTLVL register by software.

If ASTLVL contains 4, no AST is pending for the current process. If ASTLVL is less than 4, an AST is pending for the mode corresponding to the value of ASTLVL. Values of ASTLVL greater than 4 are reserved. | Execution of MTPR src, #PR$_ASTLVL with src<31:0> GEQU 5 results in UNDEFINED behavior. The preferred implementation is to cause reserved-operand fault. Processor initialization sets ASTLVL to 4. Note that loading ASTLVL with MTPR does not affect SISR or request a software interrupt. Those effects of ASTLVL occur only during REI.

6.3 PROCESS-SCHEDULING INTERRUPTS

Two of the software interrupt priorities are reserved for process scheduling software.

IPL 2 (AST delivery interrupt) is triggered by an REI that detects PSL<CUR_MOD> GEQU ASTLVL and indicates that a pending AST may now be delivered for the currently executing process.

IPL 3 (process-scheduling interrupt) is triggered by software. It indicates that a process has changed software priority and that the process scheduler should reschedule to find the highest priority executable process to run.

6.4 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. The scheduler usually runs on the interrupt stack as the result of interrupts, which are typically serviced on the interrupt stack. 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  {PCB.r*, -(KSP).w*}
```

Operation:

```
        if PSL<VM> EQLU 1 AND VMPSL<CUR_MOD> EQLU 0 then
                then {initiate VM-emulation trap};
                ! Exception frame contains no operands.
                end;

        if PSL<CUR_MOD> NEQU 0
                then {privileged instruction fault};
        if PSL<IS> NEQU 1 then {UNDEFINED};
        if {ASN IPR implemented} then
                begin
                ASN <- (PCB+96)<31:8>'0<7:0>;
                if (PCB+96)<7:0> NEQU CPUID<7:0> then
                        {invalidate per-process TB State on scalar processor
                         associated with the current value of ASN};
                end;

        else {invalidate per-process translation buffer
              entries on scalar processor};
        ! The PCB is located by the address in PCBB.
        if {internal registers for stack pointers} then
                begin
                KSP <- (PCB);
                ESP <- (PCB+4);
                SSP <- (PCB+8);
                USP <- (PCB+12);
                end;
        R0 <- (PCB+16);
        R1 <- (PCB+20);
        R2 <- (PCB+24);
        R3 <- (PCB+28);
        R4 <- (PCB+32);
        R5 <- (PCB+36);
        R6 <- (PCB+40);
        R7 <- (PCB+44);
        R8 <- (PCB+48);
        R9 <- (PCB+52);
        R10 <- (PCB+56);
        R11 <- (PCB+60);
        AP <- (PCB+64);
        FP <- (PCB+68);
        tmp1 <- (PCB+80);
        if {tmp1<1:0> NEQU 0} then {UNDEFINED};
        P0BR <- tmp1;
        if (PCB+84)<31:27> NEQU 0 then {UNDEFINED};
        if (PCB+84)<23:22> NEQU 0 then {UNDEFINED};
        P0LR <- (PCB+84)<21:0>;
        if (PCB+84)<26:24> GEQU 5 then {UNDEFINED};
        ASTLVL <- (PCB+84)<26:24>;
        tmp1 <- (PCB+88);
        tmp2 <- tmp1 + 2**23;
        if {tmp2<1:0> NEQU 0} then {UNDEFINED};
        P1BR <- tmp1;
        if (PCB+92)<30:22> NEQU 0 then {UNDEFINED};
        P1LR <- (PCB+92)<21:0>;
        PME <- (PCB+92)<31>;

        ISP <- SP;                      ! Save the interrupt stack pointe
        {interrupts off};
        PSL<IS> <- 0;                   ! Change from the interrupt stack
        SP <- (PCB);                    ! to the new kernel stack.
        {interrupts on};
        -(SP) <- (PCB+76);              ! Push PSL onto kernel stack.
        -(SP) <- (PCB+72);              ! Push PC onto kernel stack.
                                        ! (If kernel stack is inaccessibl
                                        ! or invalid, then UNDEFINED.)
```

Condition Codes:

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

Exceptions:

reserved operand
privileged instruction
VM emulation

Opcode:

06 LDPCTX Load Process Context

Description:

If the processor is in VM mode and the virtual machine is in kernel mode, then a VM-emulation trap is taken. Otherwise, if the processor is not in kernel mode, a privileged-instruction fault is taken. If neither exception is taken, the processor loads the process state in the process control block specified by the privileged PCBB register. The general registers, process-space memory management registers, and address space number register (if implemented) are loaded from the PCB into the scalar processor. 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.

If the processor implements the ASN register, the process TB state associated with the new value of ASN (the one loaded by LDPCTX) is flushed if the process last ran on a different processor, which is indicated by PRVCPU being not equal to the CPUID register. If the processor does not implement the ASN register, the process-space TB state is unconditionally flushed.

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. Other processors do not keep a copy of all four per-process stack pointers in internal registers. Rather such processors keep only the stack pointer for the current access mode in an internal register and switch this with the PCB contents whenever the current access mode changes.

2. The preferred implementation of UNDEFINED operation is reserved operand abort.

3. LDPCTX does not invalidate per-process TB state in the TB of the vector processor. To invalidate TB-state on the vector processor use the TBIA, TBIS, or VTBIA internal processor registers.

4. LDPCTX does not load the memory management registers of the vector processor, if such copies reside there.

5. \Loading ASTLVL with LDPCTX does not affect SISR or request a software interrupt. Those effects of ASTLVL occur only during REI.\

6. To guarantee correct operation, a LDPCTX must be followed by an REI instruction.

SVPCTX Save Process Context

Purpose: save register context

Format:

```
opcode  {(SP)+.r*, PCB.w*}
```

Operation:

```
if PSL<VM> EQLU 1 AND VMPSL<CUR_MOD> EQLU 0 then
        {initiate VM-emulation trap};
        ! Exception frame contains no operands.

if PSL<CUR_MOD> NEQU 0 then
        {privileged instruction fault};
!PCB is located by 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 {ASN IPR implemented} then
        (PCB+96)<7:0> <- CPUID<7:0>;
if PSL<IS> EQLU 0 then
        begin
        PSL<IPL> <- MAXU(1, PSL<IPL>);
        (PCB) <- SP;            !save KSP
        KSP <- SP;
        {interrupts off};
        PSL<IS> <- 1;
        SP <- ISP;
        {interrupts on};
        end;
```

Condition Codes:

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

Exception:

privileged instruction
VM emulation

Opcode:

```
07    SVPCTX  Save Process Context
```

Description:

If the processor is in VM mode and the virtual machine is in kernel mode, then a VM-emulation trap is initiated. Otherwise, the process control block is specified by the privileged PCBB register. 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 the processor implements the ASN register, then CPUID is saved in the PRVCPU field in the PCB. If a SVPCTX instruction is executed when the processor is executing on the kernel stack, then the processor switches to the interrupt stack and IPL is maximized with 1.

Notes:

1. The per-process memory management registers, the ASN register, ASTLVL, and PME from 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. Other processors do not keep a copy of all four per-process stack pointers in internal registers. Rather these processors keep only the stack pointer for the current access mode in an internal register and switch this with the PCB contents whenever the current access mode 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.

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 with an interrupt.

```
;
;               ENTERED WITH 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 @#PCB, #PR$_PCBB   ; Set PCB address in PCBB
                LDPCTX                  ; Load context from PCB
                                        ; For new process
                REI                     ; Place process in execution
```

Change History:

Revision J. Rich Brunner, Tim Leonard, December 1989.
- o Add ECO 119 -- which added process IDs by using the ASN IPR. ASN field put in PCB. ECO 119 revised ECO 103 which was an earlier attempt to add Process IDs.
- o Location of PCB and type of address in PCBB determined by physical address mode.

Revision H. Tim Leonard, May 1987.
- o Support virtual machines.
- o Allow PCBB to contain a virtual address.

Revision E. Al Thomas, September 1986.
- o Modify LDPCTX to include rtVAX subset.
- o Add implied operands to LDPCTX and SVPCTX.

Revision D. Tim Leonard, March 1985.
- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Display the PCB in a different format.
- o LDPCTX is UNPREDICTABLE when executed from the kernel stack.
- o Note that operation of the processor is UNDEFINED should access violation or page fault occur on the stack during LDPCTX.

Revision 6. Dileep Bhandarkar, 26 July 1982.
- o Note that ASTLVL has an effect only during REI.

Revision 5. Dileep Bhandarkar, 21 May 1980.
- o Update KSP in SVPCTX.

Revision 4, fixes to LDPCTX and SVPCTX. Peter Conklin and Ted Taylor, 10 August 1978.
- o ASTLVL<31:0> GEQU 5 is undefined.
- o Add privileged register descriptions of ASTLVL and PME, including their initial values.
- o ASTLVL GEQU 5 is reserved.
- o Contents of PCBB is initially UNPREDICTABLE.
- o Change "privileged instruction" to "reserved instruction."
- o Change R12, R13 to AP, FP.
- o Scheduler runs on IPL = 3.
- o Add software offsets for PCB.
- o Add notes to LDPCTX and SVPCTX.
- o Add reserved operand checks to LDPCTX.

Revision 3, total rewrite. Dick Hustvedt, 8 June 1976.
- o All process structure instructions described in previous revisions are deleted as well as the descriptions of firmware implemented scheduling. These changes were made as the result of efforts to simplify the VAX architecture and reduce implementation risks. The primitives described in this revision are believed to be sufficient to implement the same function in software.

- Deleted CHMI instruction.
- Changed LDPCTX to effect transition to Kernel Stack.
- Changed SVPCTX to include function of CHMI.
- Add translation buffer purge to LDPCTX.
- Changed interrupt priority level numbering to eliminate negative IPL.

Revision 2, revised for software needs. Dick Hustvedt, January 1976.
Revision 1, original issue. Tom Hastings, October 1975.

CHAPTER 7

MEMORY AND I/O

The layout and detailed structure of physical memory and I/O space is implementation dependent (and is described in Appendix B), but the overall structure of VAX memory and I/O is part of the architecture.

7.1 MEMORY, MULTIPROCESSING, AND INTERPROCESSOR COMMUNICATION.

A VAX system comprises one or more VAX processors and one or more I/O devices which all share a common memory. The VAX architecture specifies how these elements communicate with each other. It specifies how data is shared between the processors and how the processors signal each other. It includes rules for the ordering of read and write operations and interrupts and for mutual exclusion to avoid data corruption by simultaneous reads and writes from multiple processors.

There are three interrelated topics.

- Signaling and data visibility. Each processor must signal the next processor or I/O device when the next activity should begin. Each processor must also ensure that the next processor or I/O device will be able to read the new values of any shared data when the signal is received. There are three principal means for signaling and ensuring data visibility: interlocked instructions, requesting an interrupt, and writing to I/O space.

- Read and write ordering. In a uniprocessor, a read returns the value of the most recent write to the same location. In a VAX multiprocessor system, there is no time standard held in common across the several processors, so terms based on time, such as "most recent", have little utility. VAX systems use the more basic concept of the ordering of events, and particularly the ordering of read and write accesses and interrupts, to coordinate their activities.

- Mutual exclusion. Multiple processes can compete for access to shared data. Unless something prevents, more than one process may simultaneously access a shared data resource in a

> "critical section". Their mutual actions can interfere with each other producing incoherent results. That is, the required relations among the data no longer hold. Mutual exclusion protocols resolve access conflicts to critical sections so that the accessed data remain coherent.
>
> VAX hardware provides primitives for the software to implement various mutual exclusion protocols. Those hardware primitives are atomic read operations and atomic write operations for some kinds of memory references, and the "interlocked" instructions for atomic read-modify-write memory references. Software can use the hardware primitives to construct many synchronization elements such as barriers, semaphores, mutual-exclusion locks, and token passing. The interlocked instructions directly implement mutual exclusion to critical sections.

The interlocked instructions have functions involving data visibility, signaling, read and write ordering, and mutual exclusion. They are central to communication and control in a VAX multiprocessor.

A vector processor synchronizes only with its paired scalar processor as described in section 14.3.2, Scalar/Vector Memory Synchronization. To synchronize two vector processors with each other, the first vector processor synchronizes with its paired scalar processor, the two scalar processors synchronize with each other according to the rules of this chapter, and finally the second scalar processor synchronizes with its paired vector processor.

The term "I/O device" includes any I/O processor, device, controller, or adapter which has control status registers in I/O space or which can read or write the common memory.

7.1.1 Interprocessor Signaling and Data Visibility

The signals listed below assure that all observers can read the signal itself and any previous writes made by the signaling processor or I/O device. The maximum delay from a program issuing a signal until the signal takes effect in the other processors is machine dependent. If a program or I/O device does not issue one of the signals, it is UNPREDICTABLE whether another observer can read the new values of all the previously-written shared data. The signals are:

1. Any interlocked instruction with its modify operand in memory space, or any I/O device's interlocked read-modify-write sequence to memory space.

2. Any interrupt request from a processor to another processor or from an I/O device to a processor.

3. Any processor (but not I/O device) write access to I/O space, including a read-modify-write access but excluding any

vector-write access. This assures that I/O devices can observe any processor data written previously to I/O space. It also assures that a processor can't write memory, start I/O, have the I/O complete and signal a second processor, and have the second processor unable to observe the writes in memory.

\ Hardware implementation note: there are some UNIBUS, QBUS, and VAXBI devices that access memory and use a read from I/O space to initiate their actions. If these busses will be connected to a processor, the processor must make both read and write accesses from I/O space be signals. \

\ Hardware implementation note: some scheme must ensure that the signals listed above produce the correct software effect. This is a particular concern when write-buffers or write-back caches are involved. Draining a write-buffer or cache before the signal is sent will work. Implementing a "snoopy" write-buffer or cache, which monitors and responds to requests from other processors or I/O devices, can also work. Other hardware schemes are possible.\

\ Hardware implementation note: In order to recover from errors, an operating system needs to isolate them to the processor mode and process to which they apply. To accomplish the isolation, the VAX architecture recommends that internal buffers, such as write buffers and write-back caches, be drained by the initiation of an interrupt or an exception, and the REI, CHMx, LDPCTX, and SVPCTX instructions. The processor should wait for errors to be reported before changing the state of the machine otherwise errors detected during draining may be reported in the mode or context of the next process. \

There are two functions involving the console that are also interprocessor signals.

4. Processor entry to console mode as the result of a HALT (from kernel mode or from the front console panel) or an error halt.

5. Any console function that changes memory state. (Most VAX consoles do not signal that a change has been made. The changes themselves, however, are made visible.)

As an example of making data visible using an interlocked instruction, consider two processors, P1 and P2. Variable X is initialized to 0, and P2 waits for P1 to change it to 1.

```
                P1                          P2
                --                          --
                MOVB    #1,X        P$2:    CMPB    #1,X
                ADAWI   #0,Z                BNEQ    P$2
        P$1:    BRB     P$1                 HALT
```

The interlocked instruction (ADAWI) makes the new value of X visible to processor P2. When P2 observes the value of 1 take effect in location X, P2 halts. If the ADAWI instruction in P1 were not there, the new value of X might never become visible to P2, and P2 could loop forever and never halt.

7.1.2 The Ordering of Reads, Writes, and Interrupts

[This section will be added by a later ECO.]

7.1.3 Atomicity and Corruption of Shared Data

A common memory subsystem allows data to be shared among the processors and I/O devices of a VAX system. Multiple simultaneous accesses may cause some data to become "corrupted" while other "atomic" accesses keep the data intact. The following two definitions precisely describe atomicity and corruption:

- Atomicity -- The access of a shared datum in memory by one processor or I/O device is classified as atomic (indivisible) or non-atomic (divisible) with respect to an access of the same datum by another processor or I/O device. A memory access is atomic when any other memory access to the same datum may occur before or after but not during the first access. A memory access is non-atomic when another memory access to the same datum may occur before, during, or after the first access. \However, one processor's read access in the middle of another processor's atomic read access is harmless.\

- Corruption -- A shared datum accessed from memory non-atomically can become corrupted: the datum when written can become any byte-wise combination of all the data that is concurrently being written to it; and when a shared datum is read, the value read may be any byte-wise combination of its original value with any new values that are concurrently being written to it. Corruption is also known as "data incoherency" and "word tearing".

  Corruption can occur two ways: from the non-atomic accesses of two or more writers, in which case the corrupted value will remain in the memory location until some program reinitializes it; or from the non-atomic accesses of a single writer and one or more readers, in which case only the read accesses are corrupted, and subsequent read accesses will return an uncorrupted value. A datum (or its value) cannot become corrupted if it is not shared, if it is always read and written by atomic accesses, or if it is accessed by multiple readers and no writers.

To support shared data, a VAX system meets these requirements:

1. The memory ensures that the granularity of access for independent modification by any processor or I/O device is the byte. 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 1000 and 1001 contain the values 5 and 6. Suppose one processor executes INCB 1000 and another executes INCB 1001. Then, regardless of the order of execution, including effectively simultaneous execution, the final contents must be 6 and 7.

2. When executing an instruction, a VAX processor always reads and always writes certain operands from memory by atomic accesses. These operands, termed atomic operands, are the following: a byte operand, an aligned word operand, an aligned longword operand (including vector longword elements), a bit-field contained within one byte, and an aligned longword address pointed to by a displacement deferred mode or autoincrement deferred mode operand specifier. An atomic operand is the only kind of shared datum for which a VAX processor necessarily provides atomic memory access.

   \ Hardware implementation note: if a processor divides an atomic write (or read) access into two or more separate writes (or reads), then some hardware mechanism must ensure that another write or read operation does not occur to the same datum between the several parts of the atomic write (or read) access. However, one processor's read access in the middle of another processor's atomic read access is harmless.\

   A modify operand is both read and written by the same instruction. The read and the write are separate accesses to the same address. Unless the modify operand is part of an interlocked instruction (see the next section), another processor or I/O device can access the operand between the read access and the write access, even if both are atomic.

3. A processor is not required to implement any of the following as atomic operands: any datum that is not naturally aligned, quadwords (including D_floating and G_floating data and vector elements, and queue-entry headers and links),

octawords (including H_floating data), character strings, numeric strings, bit-fields not contained within one byte, packed decimal strings, or vectors. Thus, for most VAX implementations, these operands will be read or written from memory by non-atomic accesses.

7.1.4 Using Interlocks to Prevent the Corruption of Shared Data

VAX processors do not provide atomic memory access to all data. To prevent the corruption of data accessed non-atomically, readers and writers must access them in a mutually exclusive manner -- at any one time there must be either only one writer to shared data, or any number of readers. There must be neither multiple writers nor both a reader and a writer. Failure to use mutual exclusion when multiple processors or I/O devices are reading or writing with non-atomic accesses produces UNPREDICTABLE results.

Seven interlocked instructions provide the hardware primitives for software to implement various mutual-exclusion protocols. The interlocked instructions test and set a memory-resident control variable, such as a lock or semaphore, through hardware-implemented atomic memory accesses: BBSSI, BBCCI, ADAWI, INSQHI, INSQTI, REMQHI, and REMQTI. A control variable is a bit string in shared memory whose value indicates whether access to the associated shared data is allowed. Software, by using interlocked instructions, sets a control variable to any of the various values which indicate whether the shared data is accessible or inaccessible to other processors and I/O devices. BBSSI and BBCCI test and set a one-bit control variable. ADAWI maintains a count in an aligned-word control variable. The INSQHI, INSQTI, REMQHI, and REMQTI instructions use an interlock on the queue header to allow queues to be maintained consistently. The control variable for these queue instructions resides in bit <0> of the queue header.

Each of these instructions accesses a control variable using an atomic sequence of operations termed an interlocked sequence. A processor wishing to begin this sequence requests access to the interlock grain, which is an implementation-dependent region of physical memory containing the control variable. When the access has been granted, the interlocked sequence starts and the interlock grain is locked (made inaccessible) to the interlocked sequences of all other processors and I/O devices. When the sequence is completed, the interlock grain is unlocked making it accessible to the interlocked sequences of other processors and I/O devices. Thus, any two interlocked sequences performed on the same interlock grain are always atomic with respect to each other -- that is, they will always interlock. Note that interlocking occurs at the level of physical memory.

The following rules apply to interlocks:

1. Non-interlocked instructions are not necessarily blocked from reading or writing a locked memory region; whether or not they are blocked is implementation dependent.

2. The aligned longword is the basic unit of interlocking. Memory hardware ensures that interlocked sequences interlock when the starting byte addresses of their control variables map to the same aligned longword in physical memory. It is UNPREDICTABLE whether interlocked sequences will interlock if they reference control variables whose lowest-order bytes are not in the same aligned longword. The memory hardware determines the size of an interlock grain. The size may vary from a longword to all of memory and may even be discontiguous: for example, every 2**16th longword may be locked.

   For example, suppose a programmer chooses to use a word in the following way: byte 1000 will be a byte counter, and byte 1001 will contain a one-bit lock. Then, even though the control variable for an ADAWI is an entire word, the programmer must not expect that a BBSSI to byte 1001 will interlock with an ADAWI to the word since only byte 1000 need be locked for the ADAWI and byte 1001 for the BBSSI.

3. It is possible in VAX implementations that interlocked sequences on two different control variables will unintentionally interlock because both control variables lie within the same interlock grain. For example, if the size of the interlock grain for a VAX processor is one page (512 bytes), then an interlocked sequence performed on a control variable within a page will lock out all other interlocked sequences which are performed on the same page.

4. A processor may read more bytes than necessary when reading a control variable for an interlocked sequence, but it only writes back the following: for BBSSI and BBCCI, the byte that contains the control variable; for ADAWI, the aligned word that contains the control variable; and for the interlocked queue instructions, the aligned longword that contains the control variable.

5. If software accesses a control variable with an interlocked instruction, then software must write the control variable only with interlocked instructions. If a non-interlocked write occurs while an interlocked sequence is in progress on the control variable, the data may be overwritten when the interlocked control variable is rewritten. Thus, when a control variable is tested or modified, it is UNPREDICTABLE whether a non-interlocked instruction updates any byte contained within a range of bytes that is written.

   For example, suppose the following: the initial contents of a word-sized control variable, CTRL, are ^X0000. Processor 1 executing (ADAWI #1, CTRL) begins an interlocked sequence on CTRL as Processor 2 executing (MOVB #8, CTRL+1) starts to write to CTRL+1. Then, when both instructions complete, the final value of CTRL could be either ^X0001 or ^X0801.

Two good-programming practices result from the above requirement.

- o If you use interlocked instructions to set or clear flag bits in a byte, use interlocked instructions for all writes to that byte.

- o If you use an interlocked instruction to allocate a resource, use an interlocked instruction to deallocate it.

6. No interlocked sequence is ever performed on a GPR. Therefore BBSSI, BBCCI, and ADAWI control variables which need to be interlocked must not be in GPRs.

7. ADAWI is the only interlocked instruction whose control variable can be located in I/O space, in which case it must also be in Unibus space. When it is in I/O space but not Unibus space, it is implementation-dependent whether an interlocked sequence is ever performed on the control variable. See section 7.5.3, Instructions Usable to Reference I/O Space.

8. For performance reasons, software should avoid looping with an interlocked instruction when waiting for access to shared data. Recall that non-interlocked instructions are not necessarily blocked from reading control variables in a locked interlock grain. On most VAX processors, once an interlocked instruction determines that shared data is not available, repeated checking for availability is most efficiently done with non-interlocked instructions. If the non-interlocked instruction indicates the data is available, then an interlocked instruction can confirm the result; otherwise, the program can continue looping on the non-interlocked instruction or do something else. Note that another processor can gain control of the data in the time between the non-interlocked and interlocked checks for data availability. For further details and an example, see the instruction descriptions of BBSSI and BBCCI in Chapter 3.

   \ Hardware implementation note: when implementing BBSSI, BBCCI, or the interlocked queue instructions, hardware designers may test the bit using non-interlocked reads. If the bit is found to be set (or clear for BBCCI), hardware can set condition codes or branch and terminate the instruction; otherwise, hardware can begin the actual interlocked sequence. (In either case, an interlocked instruction must accomplish the data-visibility side effects described in section 7.1.1.)

However, if the non-interlocked read finds the lock available, the cost of obtaining the lock may be twice that of performing the interlocked read first. That can occur if the non-interlocked read misses in the processor cache. Then two memory reads are required to obtain the interlock, one for the non-interlocked read that missed, and one for the interlocked read that must follow.

To avoid the above problems entirely, hardware designers should strongly consider adding mechanisms to avoid deadlocks when multiple processors and I/O devices are simultaneously attempting interlocked sequences. \

7.2 INSTRUCTION STREAM PROCESSING AND MODIFICATION

This section briefly describes the processing of instruction streams in the VAX Architecture and how to ensure correct processing behavior when the instruction stream is modified.

The VAX Architecture assumes a sequential control flow model of instruction stream processing. In this model, instructions are processed from the instruction stream, which resides in instruction memory, one at a time and no instruction is processed (even partially) until the one before it completes.

An instruction consists of its opcode, all operand specifiers, and any operands that are necessarily contiguous with the rest of the instruction stream. Thus, the instruction stream includes short literals, immediate-mode operands, absolute addresses in absolute mode addressing, branch displacements, CALLx entry masks, and CASEx tables. Except for immediate-mode operands and branch displacements, the instruction stream does not include operands, even ones addressed relative to the PC. Except for absolute addresses, the instruction stream does not include indirect addresses. It also does not include EDITPC patterns. All instruction operands and indirect addresses not considered to be within the instruction stream are considered data.

Instruction memory is those portions of memory used for the storage and retrieval of instructions to be executed by the processor.

No portion of instruction memory may reside in page tables, PCBs, or the SCB. Instructions also may not lie within the following region above the top-of-stack as the region is allowed to change in an unpredictable manner: the region starts at -1(SP) and ends at either -5*512(SP) or the first non-writable page (within the current mode) above the top-of-stack, whichever comes first. The operation of the processor is UNPREDICTABLE when it executes instructions retrieved from the above mentioned regions.

The processing of one instruction entails translating the instruction address using the appropriate page table entry; fetching the instruction from instruction memory with the translated address;

decoding the instruction and fetching required operands; executing the instruction; and storing the results, if any, in the indicated locations. Thus, the processing of an instruction stream requires various resources such as instruction memory, page tables, general purpose registers, and privileged registers.

7.2.1 Prefetching of Instructions

Since a strict hardware implementation of the sequential control flow model yields poor performance, many VAX processors overlap the processing of instructions to increase their speed. This performance improvement is possible by utilizing instruction lookahead and buffering logic, Translation Buffers, and multiple processors. Instruction lookahead and buffering logic includes but is not limited to instruction buffers, virtual instruction caches, and physical instruction and data caches. Note that in utilizing these techniques, VAX processors are still generally required to maintain to software the sequential control flow model of instruction stream processing. However VAXes are allowed to deviate from this model during instruction stream modification (which is discussed in the next section) when changes to instruction memory do not necessarily update a processor's prefetched copy of the same.

7.2.2 Instruction Stream Modification

An instruction stream is modified whenever its contents are changed, added to, or deleted. This modification can occur when changing any of the resources used in determining the location, content, or mapping of the instruction stream (listed in Table 7-1). For example, writing data into instruction memory, changing the value of a valid PTE, or changing memory management mode modifies the instruction stream.

However, changing any of the above resources does not always modify the instruction stream. For example, decrementing P0LR by 4 to exclude an invalid PTE at the end of the P0 page table does not modify the instruction stream. The determination of which changes to the above resources actually cause instruction stream modification is often complex; therefore, it is recommended that software treat any change to these resources as causing an instruction stream modification, if there is any doubt.

For VAX processors which implement lookahead and buffering logic, instruction stream modification can cause the processor's buffered copy of the instruction stream to become stale -- that is, no longer match the modified instruction stream in instruction memory. From the processor's point of view, the modification has created two equally valid instruction streams (the original one and the modified one) from which the processor can choose to fetch instructions. As a result, the processor may execute parts of both the unmodified and modified instruction streams. This situation can go on indefinitely unless the processor is forced by software to stop using stale copies of the instruction stream.

Software can force the processor to stop using stale copies of the instruction stream and begin to fetch instructions only from the modified instruction stream by using the REI instruction. To ensure correct results, whenever software modifies the instruction stream by changing any of the resources listed in Table 7-1 on a given processor, software must issue an REI on that processor after the change and before the processor actually uses (as seen by software under a sequential control flow model) the changed resource to process the modified instruction stream. Furthermore, software must flush stale PTEs from that processor's translation buffer before issuing the REI if the changed resource pertains to memory management. (This TB flush requirement can be waived during context switching if the TB entries are valid for the previous process and the hardware supports the association of TB entries with ASN values.)

Until software issues the REI and flushes stale PTEs from the translation buffer (when appropriate), it is UNPREDICTABLE whether the modified or unmodified instruction stream or bytes of each will be executed by the affected processor. Note that if software changes the next instruction location to be executed in the instruction stream, it is always UNPREDICTABLE whether the old or new instruction in that location will be executed.

If the changed resource is used immediately after the instruction stream modification for the processing of subsequent instructions, then software must ensure that these instructions and the required REI reside in the same locations in both the unmodified and modified instruction streams. For example, if software changes the mapping for the current page of the instruction stream by writing into the page's PTE, then software can force the processor to use the updated instruction memory by guaranteeing that the next two instructions after the PTE write are an MTPR-to-TBIS (to flush the old PTE value out of the TB) followed by REI. It does this by placing an MTPR-to-TBIS and an REI in the next locations to be executed by the processor under either the old or new PTE mapping.

In the case of a multiprocessor configuration, software must apply the above rule to each processor in turn. Software must flush stale PTEs from the TB and issue an REI on each processor whose instruction stream has been modified.

Note that flushing PTEs from the TB by writing to TBIA or TBIS does not modify the instruction stream since the PTEs in the TB are merely copies of the PTEs in memory. Thus, instructions fetched using a particular TB entry are still valid after the TB entry is flushed as long as the underlying PTE in the page table is unmodified. If this PTE in memory is modified, then the process state is changed and software must issue an REI instruction as described earlier in this chapter and in Table 7-1.

Executing an REI on most VAX processors is a time consuming operation. To minimize execution time, software should avoid issuing excessive REIs. When multiple instruction stream modifications are necessary, software should perform them all first and then issue a single REI afterwards. For example, if a debugger is inserting BPT instructions into the instruction stream, it should do all necessary changes and then issue an REI when finished.

Lastly, the above rules do not apply to a compatibility mode procedure, which can modify its instruction stream without any additional synchronization.

HARDWARE IMPLEMENTATION NOTE

> \ When any instruction processing resource described in Table 7-1 is changed, hardware is not required to flush instruction lookahead or buffering logic until software issues an REI. Furthermore, as described above, the processor is not required to flush the instruction lookahead or buffering logic when TBIA or TBIS is written. \

Table 7-1: Resources that when changed can modify the Instruction Stream

Resource	How Changed	Software Action Necessary After The Change When Instruction-Stream Modification Occurs
Instruction Memory	Writes	Issue REI
Valid Process PTEs	Writes	If MME=1 then Flush TB[Px] and Issue REI.
PxBR, PxLR,	MTPR	If MME=1 then Flush TB[Px] and Issue REI.
ASN	MTPR	If MME=1 then Issue REI.
PxBR, PxLR, ASN	LDPCTX	Issue REI
Valid System PTEs	Writes	If MME=1 then Flush TB[sys] and Issue REI.
SBR, SLR	MTPR	If MME=1 then Flush TB[sys] and Issue REI.
MAPEN	MTPR	Issue REI*.

Writes = Writes to instruction operands (explicit or implicit) in memory or memory writes by I/O or another processor.

REI = Software must issue an REI after the instruction stream has been modified and before the modified instruction stream is processed (as seen by software under a sequential control flow model) using the changed instruction processing resource.

Flush TB[Px] = Software must flush stale process PTEs from the TB.

Flush TB[sys] = Software must flush both stale system and process PTEs from the TB.

* = Before enabling MAPEN, software must flush the entire TB. Note that an REI should always be issued after changing memory management mode. \The REI after enabling MAPEN is not immediately necessary if the operating system chooses to handle the two instruction-streams at once model. For example, when VMS enables MAPEN, VMS does not immediately issue an REI because it has properly ensured that until it issues the REI, each page used by VMS either has an identical physical and virtual address or the page's virtual address is outside the physical address range when MAPEN=0.\

7.3 MEMORY REFERENCES

The memory references made by each instruction (and therefore the possible memory exceptions) are specified as part of the VAX architecture. Any required or permitted memory reference (read, modify, or write) may be made more than once, except for references to I/O space which are made once and only once. Operands requiring interlocked access are always referenced. In general, for operands not requiring interlocked access, it is UNPREDICTABLE whether an operand is referenced if it does not affect the result (including condition codes). Further clarifications and exceptions to this simplified rule are listed below. Software must not rely on the occurrence of memory management exceptions on operands that do not affect the result of an instruction. The probe instructions should be used to determine the accessibility of a memory location. In general, no results are written unless the instruction can be completed or can be suspended with PSL<FPD> set (but see section 5.7.5, Memory Above Top-of-Stack).

1. It is UNPREDICTABLE whether longwords containing indirect addresses are read. For example, MULL3 #0, @16(R5), A may or may not access the longword containing the address of the second operand.

2. If a branch is not taken, it is UNPREDICTABLE whether the branch displacement is read.

3. It is UNPREDICTABLE whether all bytes for .r operands are read. For example, TSTF may only read the word containing the sign and exponent. BLBC and BLBS may only read the low byte of the source operand.

4. All bytes for .w operands are always written. \There is no way to finish an instruction without writing the result, except for cases like MOVL A, A and MULL3 A, #1, A.\

5. It is UNPREDICTABLE whether all bytes for .m operands are either read (with modify intent) or written. However, a modify operand requiring interlocked read and write is always accessed. For example, ADDL2 #0, A may only read A (without modify intent). INCL A may only write the bytes of A that changed. The sum operand of ADAWI #0, A is always read and written back interlocked.

6. For .a operands (and for .v operands when .v is not a register), the memory reference behavior is peculiar to each instruction or instruction group. Overriding the rules given below, it is UNPREDICTABLE whether an otherwise unreadable operand is read or not if it appears as an immediate mode operand. For example, PUSHAB (R0) cannot read the byte at (R0), but PUSHAB #512 can read the value 512.

   a. POLY{F,D,G,H}. If the argument is not zero, each entry in the coefficient table is read unless an arithmetic exception occurs before the instruction completes. If the argument is zero, it is UNPREDICTABLE whether the entire table or only the last coefficient is read.

b. MOVA{B,W,L,Q,O} and PUSHA{B,W,L,Q,O}. The address operand is not referenced.

c. Field Instructions (EXTV, EXTZV, INSV, CMPV, CMPZV, FFS, FFC). The aligned longword(s) containing the field specified by FIELD (pos, size, base) can be read. For INSV, only this aligned longword(s) can be written. It is UNPREDICTABLE whether all or some of the bytes in these longwords are accessed. Furthermore, no memory references occur for a zero-length bit string.

d. BB{S,C}, BB{S,C}{S,C}. Only the single byte containing the test bit specified by the base and position operands is read. If the test bit does not need to change state, it is UNPREDICTABLE whether the byte is written back.

e. BB{SS,CC}I. Only the single byte containing the test bit specified by the base and position operands is referenced using the interlocked forms of read and write. The test bit is written even if its state is unchanged.

f. JMP and JSB. The address is not referenced by the JMP or JSB (but will be read as instruction stream data for the next instruction).

g. CALL{S,G}. The two bytes (containing the entry mask) at the destination address are read. The argument list for CALLG is not referenced.

h. Interlocked Queue. It is UNPREDICTABLE whether the backward pointer of the queue header is accessed for INSQHI, REMQHI.

i. Character String instructions (CMPC, LOCC, MATCHC, MOVC, MOVTC, MOVTUC, SCANC, SKPC, SPANC). No memory references occur for a zero-length character string.

j. Decimal instructions (ADDP, ASHP, CMPP, CVTLP, CVTPL, CVTPS, CVTPT, CVTSP, CVTTP, DIVP, MOVP, MULP, SUBP). No memory references occur for a zero-length trailing numeric string.

7. Some of the character string instructions (MOVTUC, CMPC3, CMPC5, SCANC, SPANC, LOCC, SKPC, and MATCHC) can stop before the whole source string is processed. Three definitions help define the required memory references for these instructions.

The stop byte is the byte that ends the instruction execution without using the string length end condition. It is the last byte on which the answer of the instruction depends. (The stop byte may have any position in the string, including first or last, or it may not exist at all. For string matches, it is the last byte of the matched string.)

A source string consists of a body concatenated with a tail.

The body of a source string is the substring from the first byte up to and including the stop byte, if one exists, or up to and including the last byte (as determined by the source string's length) if no stop byte exists. (The body may be null only if the source string has a zero length.)

The tail of a source string is the substring from the first byte after the body up to and including the last byte in the source string as determined by the source string's length. (The tail will be null if there is no stop byte or if the stop byte is the last byte.)

Character strings are defined by length and starting address. Some strings (ASCIZ strings) are delimited by a specific character. The "real" length of the string is not known, and 64K is used as the length. Only some of the VAX character string instructions can be reasonably used on character delimited strings. These instructions are MOVTUC, SPANC, SCANC, LOCC, and SKPC. For these five instructions, it is necessary to guarantee that no memory management exceptions will occur beyond the page containing the delimiting character. The absence of such a requirement could cause a program that works on one processor to fail on another because of access violations on data that is not necessary to produce the correct result.

For string operands specified by length and starting address, one of the following rules applies:

a. For MOVC3, MOVTC, and CRC, all bytes are referenced. These instructions have no end condition other than string length.

b. For MOVC5, the stop byte is defined as the last byte moved from the source string. MOVC5 references all bytes except when the source string is longer than the destination string; in the latter case, no bytes in the source string's tail beyond the page containing the stop byte are referenced.

c. For CMPC3, CMPC5, and MATCHC, all bytes in a string's body are referenced. It is UNPREDICTABLE whether any bytes in a string's tail are referenced.

d. For MOVTUC, SCANC, SPANC, LOCC, and SKPC, all bytes in the source string's body are referenced, and no bytes in the source string's tail beyond the page containing the stop byte are referenced. For MOVTUC, the destination address which would receive the translated escape character is not written into, nor is any larger address written into.

For table operands, one of the following rules applies:

a. In the table for MOVTC, MOVTUC, SCANC, and SPANC, entries are accessed for the corresponding source characters or values. It is UNPREDICTABLE whether the other table entries are accessed.

b. For the CRC table operand, it is UNPREDICTABLE whether all or only part of the table is accessed.

8. If a packed decimal source string contains invalid digits, it is UNPREDICTABLE whether the entire source string is read and whether any or all of the destination is written.

   If there are no invalid digits in a packed decimal source string, one of the following rules applies:

   a. EDITPC, MOVP, ADDP6, SUBP6, MULP, DIVP, CVTPT, CVTTP, CVTPS, CVTSP, and ASHP. All bytes of the source strings are read, and all bytes of the result are written, unless an exception condition is detected and the instruction can be completed without reading all the bytes in the source strings.

   b. CMPP3 and CMPP4. It is UNPREDICTABLE whether all bytes of the two source strings are read.

   c. ADDP4 and SUBP4. All bytes of the addend (or subtrahend) string are read. It is UNPREDICTABLE whether all bytes of the result are written.

   d. CVTLP. All bytes of the destination string are written.

   e. CVTPL. All bytes of the source string are read.

   f. EDITPC, CVTPT, CVTTP. The table entries are accessed for the corresponding source bytes. It is UNPREDICTABLE whether the other table entries are accessed.

9. PROBER and PROBEW. The first and last bytes specified by the base and length operand are not accessed.

7.4 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, and recovery). In particular, the following must be true:

1. An I/O transfer from memory to a peripheral, started after a program write to the same memory, must output the updated memory value.

2. A program memory read, executed after the completion of an I/O transfer from a peripheral to the same memory, must read the updated memory value.

3. If one processor writes or modifies memory and then executes HALTs, a read or modify of the same memory by another processor must read the updated value.

4. If a processor writes or modifies memory and then halts as a result of power failure, a read or modify of the same memory 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.

7. A cache may prefetch instructions or data. In a virtual cache, memory management exception conditions could occur during prefetch. Such exceptions should not be taken until the prefetched data is referenced by an instruction.

Processor initialization must leave the cache either empty or valid.

7.5 I/O STRUCTURE

The VAX I/O architecture is very similar to the PDP-11 structure. The principal difference is the method by which internal processor registers (such as the memory management registers) are accessed. Peripheral device control and status registers and data registers appear at locations in part of the physical address space (I/O space), and can therefore be manipulated by most memory reference instructions. Use of general instructions permits all the virtual address mapping and protection mechanisms described in Chapter 4 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. The physical address space consists of two parts: memory space and I/O space. Memory space starts at address zero and continues to an implementation-dependent limit. I/O space begins at that limit and continues to the end of the physical address space.

For any member of the VAX series implementing the UNIBUS interconnect, there will be one or more areas of I/O space, each 2**18 bytes in length, that "map through" to UNIBUS addresses. The collection of these areas is referred to as the UNIBUS space.

\Hardware Implementation Note. There are some VAXBI device control registers that require word accesses, that is, they decode A<1> and/or use a mask for word write-accesses. These devices are usually architecturally derived from UNIBUS devices. Among them are the KDB50 and the the KLESI-B (TU81 and RV_WORMs). There are also an unknown number of third-party VAXBI devices which require word accesses. Although there are no known VAXBI devices which require byte accesses, even this is possible. VAX processor designers should build VAX hardware to make word I/O accesses unless the project plan specifically excludes VAXBI requirements. To fully support both word and byte accesses means that: the entire VAXBI address, including A<1:0>, must correspond to the specifier for all VAXBI transactions; and the four mask bits must correspond to the specifier and the instruction type for write-type VAXBI transactions.

Similarly, there are XMI device control registers that require word accesses. Therefore, the eight XMI mask bits must correspond to the specifier and instruction type for write-type XMI transactions unless the project plan specifically excludes XMI requirements. \

7.5.1 I/O-Device Use of PTEs

\Note: This section has no impact on processor implementations. The PTE formats described are for use only by software and hardware using the CI bus.\

Some I/O devices, such as those using the VAX CI Port Architecture, use VAX page tables to translate addresses. These I/O devices use the PTE formats described in Chapter 4, Memory Management, including particular field encodings of invalid PTEs. Specifically, I/O devices for VAX processors with 34-bit physical addresses decode PTE<31,26,25> and use a 25-bit page frame number (PFN); I/O devices for VAX processors with 30-bit physical addresses decode PTE<31,26,25> (or PTE<31,26,22> for older devices) and use a 21-bit PFN. The PTE bits are decoded into four combinations, as shown in Table 7-1. Some of these are used in the same way as in the VAX-processor PTE formats, and some are used in different ways.

When PTE<31> = 1, on systems with 34-bit physical addresses PTE<24:0> is a 25-bit PFN, and on systems with 30-bit physical addresses PTE<20:0> is a 21-bit PFN. These two formats are shown in Figure 7-1a. PTE<PROT> and the PFN fields are identical to those used by VAX processor hardware, and they are used the same way. The mapped pages may be used by either the processor or I/O devices.

When PTE<31,26,25> = 000 (binary), on systems with 34-bit physical addresses PTE<24:0> is a 25-bit PFN, and on systems with 30-bit physical addresses PTE<20:0> is a 21-bit PFN. On 30-bit systems, PTE<22> is a copy of PTE<25> so that older 30-bit I/O devices can decode PTE<31,26,22> = 000 (binary) to get the 21-bit PFN. These formats are shown in Figure 7-1b. The formats are not used by VAX processor hardware. I/O devices can use the formats only if they have established a cooperating protocol with the operating system. Then the operating system will guarantee that the mapped pages remain memory resident as long as the I/O is still operating.

When PTE<31,26,25> = 001 (binary), PTE<21:0> is a 22-bit global page table index (GPTX). On 30-bit systems, PTE<22> is a copy of PTE<25> so that older 30-bit I/O devices can decode PTE<31,26,22> = 001 (binary) to get the 22-bit GPTX. This format is shown in Figure 7-1c. The format is not used by VAX processor hardware. I/O devices can use the format only if they have established a cooperating protocol with the operating system. The I/O device has a global-page-table-base register (GBR) that is loaded (via memory) 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 have a valid PFN and must have its decoded bits equal to either 1xx or 000 (binary). If either of these requirements is not met, the result is UNDEFINED.

When PTE<31,26> = 01 (binary), as shown in Figure 7-1d, the PTE format is reserved to DIGITAL. I/O devices will abort in a device-dependent manner.

I/O devices which decode PTE<31,26,25>, either 34-bit or 30-bit devices, never look at or check PTE<PROT> or modify PTE<M>. I/O devices which decode PTE<31,26,22>, that is, older 30-bit devices, may (it's device dependent) look at and check PTE<PROT>, but they never modify PTE<M>.

The I/O devices described in this section do memory mapping using the same system page table as the VAX processor, but they have their own copies of the processor's SBR and SLR which the software loads by writing to memory.

I/O-device 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. \Process virtual addresses are not passed directly to I/O devices because the software needs to further qualify them and specify an associated P0BR and P1BR. This scheme simplifies the I/O hardware design and incurs no unnecessary software overhead. \ In addition, the I/O devices use a global page table in memory and an I/O hardware global-page-table-base register which the software loads by writing to memory.

Table 7-2: PTE Types For Use By I/O

PTE Type	PTE<31,26,25,22>			
Processor-valid PFN	1	x	x	x
Resident-for-I/O PFN	0	0	0	0*
Global Page Table Index	0	0	1	1*
I/O abort	0	1	x	x

* PTE<22> is a copy of PTE<25> for older 30-bit I/O devices using 21-bit PFNs.

```
 3  3      2 2  2  2   2 2  2  2
 1  0      7 6  5  4   3 2  1  0                                      0
+--+--------+--+--+-----+--+--+----------------------------------------+
| 1|  PROT  | M| S|  S  | S| S|           21-bit PFN                   |
+--+--------+--+--+-----+--+--+----------------------------------------+

+--+--------+--+--+----------------------------------------------------+
| 1|    S   | S| S|                       25-bit PFN                   |
+--+--------+--+--+----------------------------------------------------+
```

a. PTEs with Processor-valid Page Frame Number. PTE<31> = 1.

```
 3  3      2 2  2  2   2 2  2  2
 1  0      7 6  5  4   3 2  1  0                                      0
+--+--------+--+--+-----+--+--+----------------------------------------+
| 0|  PROT  | 0| 0|  S  | 0| S|           21-bit PFN                   |
+--+--------+--+--+-----+--+--+----------------------------------------+

+--+--------+--+--+----------------------------------------------------+
| 0|    S   | 0| 0|                       25-bit PFN                   |
+--+--------+--+--+----------------------------------------------------+
```

b. PTEs with Resident-for-I/O Page Frame Number. PTE<31,26,25> = 000.

Figure 7-1 PTEs For Use By I/O Devices

```
 3  3      2 2  2  2   2 2  2
 1  0      7 6  5  4   3 2  1                                         0
+--+--------+--+--+-----+--+-------------------------------------------+
| 0|  PROT  | 0| 1|  S  |1*|              22-bit GPTX                  |
+--+--------+--+--+-----+--+-------------------------------------------+
```

* PTE<22> is a 1 for devices which decode PTE<31,26,22>, that is, older devices using 21-bit PFNs. Otherwise it is "reserved for software".

c. PTEs with Global Page Table Index. PTE<31,26,25> = 001.

```
 3  3      2 2  2
 1  0      7 6  5                                                     0
+--+--------+--+-------------------------------------------------------+
| 0|  PROT  | 1|              reserved for software use                |
+--+--------+--+-------------------------------------------------------+
```

d. I/O Abort. PTE<31,26> = 01.

Figure 7-1 PTEs For Use By I/O Devices (cont)

7.5.2 Restrictions 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); that is, all registers must be aligned on natural boundaries.

2. References using a length attribute other than the length of the register, or to unaligned addresses, 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 are not affected by writing a 0. This is to prevent clearing bits that may be asynchronously set between reading and writing a register.

4. The following applies only to systems that support a UNIBUS. Only byte and word references of read-modify-write type (.mb or .mw access type) in UNIBUS I/O spaces are guaranteed to interlock correctly. Processors must ensure that .mb and .mw access types use the DATIP - DATO(B) functions when the operand physical address selects a UNIBUS device. This constraint does not apply to longword, quadword, field, all floating, or string operations if implemented using byte- or word-modifying references. This constraint also does not apply to instructions precluded from I/O space references. References in the I/O space other than in UNIBUS spaces are UNDEFINED with respect to interlocking. This includes the BBSSI and BBCCI instructions.

5. String, quadword, octaword, F_floating, D_floating, G_floating, H_floating, and field references in the I/O space result in UNDEFINED behavior.

6. Page tables must not be located in I/O space. References to page table entries located in I/O space result in UNDEFINED behavior.

7. The PCB and SCB must not be located in I/O space. References to the PCB or to SCB entries located in I/O space result in UNDEFINED behavior.

8. Unless otherwise stated as part of the device specification, a read from a device register after a write will return any values updated by the write. \ This requires that all involved hardware (processors, bus adapters, and devices) process reads and writes in the order issued by the program. \

9. Software must not depend on a write to I/O space completing with the instruction doing the write: a processor may continue instruction execution without waiting for the written data to reach its destination. The mechanism for ensuring the currency of the effects of a write is a read from the same device. (Note that I/O-space reads are much slower than writes since the processor must wait for round-trip signals from the I/O device.)

10. The time for an interrupt to occur after a program initiates device actions resulting in interrupts is UNPREDICTABLE. Also, the time for the last interrupt to occur after a device's interrupts have been disabled is UNPREDICTABLE. A program can set a software flag indicating that any interrupts for a device should be ignored.

11. The time for an I/O-system error to be reported is UNPREDICTABLE. Therefore the PC when the error report occurs can be far removed from the instruction that caused the error. \ An operating system needs to determine for which device an error occurred in order to limit the consequences of the error. The unpredictable time makes the determination difficult or impossible. \

12. I/O devices that reference memory are always started by a write to I/O space and never by a read. This assures that any data written previously (by the processor starting the I/O device) may be read by the device. (See section 7.1.1, Interprocessor Signaling and Data Visibility.)

7.5.3 Instructions Usable to Reference I/O Space

Some instructions must not be used for referencing I/O space. Reasons for this include:

1. String instructions are restartable with PSL<FPD>.

2. The PC, SP, or PCBB cannot point to I/O space.

3. 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.

4. The instruction may be interruptible because it is potentially a slow instruction in some implementations.

5. Only instructions with a maximum of one modify or write destination can be used. The destination must be the last operand.

Vector instructions are not allowed to reference I/O space. If a vector instruction references I/O space, the results are UNPREDICTABLE. \ The preferred, but not required, implementation is to generate an access control violation with the VIO bit of the memory management fault parameter set to distinguish the access control violation as an illegal vector I/O space reference. \

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 exceptions will occur, including page fault, modify fault, or overflow trap, after the first I/O space reference. To ensure no interrupts, the programmer must avoid operand specifier modes 9, 11, 13, and 15, and these modes indexed. (Symbolically, these are @(Rn)+, @B^D(Rn), @W^D(Rn), and @L^D(Rn), and these indexed.) The hardware may allow interrupts for these modes in order to minimize interrupt latency. For the instructions in the following lists, the hardware ensures that no other interrupts will occur after the first I/O space access.

Since these instructions are not interruptible 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:

ADAWI
ADD{B,W,L}2
ADD{B,W,L}3
ADWC
BIC{B,W,L}2
BIC{B,W,L}3
BICPSW
BIS{B,W,L}2
BIS{B,W,L}3
BISPSW
BIT{B,W,L}
CASE{B,W,L}
CLR{B,W,L}
CHM{K,E,S,U}
CMP{B,W,L}
CVT{BW,BL,WB,WL,LB,LW}
DEC{B,W,L}
INC{B,W,L}
MCOM{B,W,L}
MFPR
MNEG{B,W,L}
MOV{B,W,L}
MOVA{B,W,L}
MOVAQ
MOVPSL
MOVZ{BW,BL,WL}
MTPR
PROBE{R,W}
PUSHA{B,W,L}
PUSHAQ
PUSHL
SBWC
SUB{B,W,L}2
SUB{B,W,L}3
TST{B,W,L}
XOR{B,W,L}2
XOR{B,W,L}3

NOTE

\If the sum operand of ADAWI is in I/O space outside UNIBUS space, it is UNPREDICTABLE whether it is accessed with an interlock.\

Instructions for which some operand can be in I/O space are as follows:

```
        BLB{S,C} (any operands but branch displacement)
        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 or to allow the stack or PCB to be in I/O space. This might, for example, be used as part of the bootstrap process. If this is done, then it is valid for software to transfer to this code.

\For reference, instructions were discarded as follows:

1. String Instructions in Base Instruction Group: MOVC3, MOVC5, CMPC3, CMPC5, SCANC, SPANC, LOCC, SKPC

2. Floating-Point Instructions in Base Instruction Group: MOV{F,D,G}, MNEG{F,D,G}, CVT{B,W,L,F,D,G}{F,D,G}, CVT{F,D,G}{B,W,L,F,D,G}, CVTR{F,D,G}L, CMP{F,D,G}, TST{F,D,G}, ADD{F,D,G}2, ADD{F,D,G}3, SUB{F,D,G}2, SUB{F,D,G}3, MUL{F,D,G}2, MUL{F,D,G}3, DIV{F,D,G}2, DIV{F,D,G}3

3. Extended Accuracy Group: MOVH, MNEGH, CVTH{B,W,L,F,D,G}, CVT{B,W,L,F,D,G}H, CMPH, TSTH, ADDH2, ADDH3, SUBH2, SUBH3, MULH2, MULH3, DIVH2, DIVH3, CVTRHL, MOVO, CLRH (CLRO), MOVAH (MOVAO), PUSHAH (PUSHAO)

4. Packed-Decimal-String Group: MOVP, CMPP3, CMPP4, ADDP4, ADDP6, SUBP4, SUBP6, MULP, DIVP, CVTLP, CVTPL, CVTPT, CVTTP, CVTPS, CVTSP, ASHP

5. Emulated Only Group: EMOD{F,D,G,H}, POLY{F,D,G,H}, ACB{F,D,G,H}, MATCHC, MOVTC, MOVTUC, CRC, EDITPC

6. PC, SP, PCBB not in I/O space and instruction has no other operands: Bxxx, BRB, BRW, JMP, BSBB, BSBW, JSB, RSB, RET, BPT, REI, HALT, NOP, LDPCTX, SVPCTX

7. operand types: MOVQ, CLRQ, ASHQ, INSQUE, INSQHI, INSQTI, BB{S,C}, BB{S,C}{S,C}, BB{SS,CC}I

8. slow: MUL{B,W,L}2, MUL{B,W,L}3, EMUL, DIV{B,W,L}2, DIV{B,W,L}3, EDIV, ASHL, ROTL, EXTV, EXTZV, INDEX, INSV, CMPV, CMPZV, FFS, FFC, CALLG, CALLS, PUSHR, POPR

9. modify or write operand must be last: ACB{B,W,L}, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR

10. Compatibility mode: all references

\

Change History:

Revision J. Rich Brunner, Tom Eggers, December 1989

- Removed references to SPTEP and SPTs in virtual memory.
- ECO 118: Replace section 7.1 with completely new section on "Memory, Multiprocessing, and Interprocessor Communication". This section addresses interlocked instructions as did the section it replaced.
- ECO 122: Updated section on I/O use of PTEs for PTEs with 25-bit PFNs. Deleted section 7.5.1 on "Mapping I/O Into Larger Addresses".
- ECO 123 -- Make the UNIBUS interconnect optional. All references to UNIBUS are amended.
- VAXBI and XMI devices require word accesses. Details in section 7.5 on I/O Structure.
- Requirement for UNIBUS DATIP - DATO(B) functions moved from old section 7.1 to section 7.5.3, Restrictions on I/O Registers.
- ECO 115A and 109: -- Instruction Stream processing and Modification becomes new section 2, replacing Separation of Procedure and Data
- Add description of behavior on zero-length operands in section 3.
- ECO 101 -- Allow writes above top-of-stack.
- ECO 95 -- Allow write-and-run to I/O space
- Fix bit numbering above register figures.
- Vector instructions not allowed to reference I/O space.

Revision H. Tim Leonard, May 1987.

- Describe mapping I/O into large physical-address spaces.
- Remove references to Kernel Instruction Set.

Revision E. Al Thomas, September 1986

- Move sections dealing with interrupts, errors, and restartability to the Exceptions and Interrupts Chapter.

Revision D. Tim Leonard, March 1985.

- Change the revision number to correspond to DEC Standard 032 rev number.
- Move to here the section from Appendix F about Instructions Usable to Reference I/O Space.
- It's always allowed to read an immediate mode operand.
- PCB and SCB must not be in I/O space.
- Overflow trap is not allowed after an I/O space reference.

Revision 6, add programming considerations. Dileep Bhandarkar, 26 July 1982.

- Add specification on required and allowed memory references.
- Move Separation of procedure and data from Chapter 2.
- Change chapter name to include programming implications.

Revision 5, I/O space access. Tom Eggers, 17 June 1980.

- I/O Space Access.

Revision 4, document cache and interlocks. Peter Conklin and Ted

Taylor, 10 August 1978.

- o Move I/O here from chapter 9.
- o Document interlocks on ADAWI.
- o Document cache and its constraints.
- o REMQUE and INSQUE are aligned (ECO).
- o Ensure that fault when interlocked does not hang.
- o Rule for updating system PTEs in a multi-processor.
- o Clean up discussion of interlocks.

Revision 3, original version. Dave Rodgers, 2 June 1976.

CHAPTER 8

PRIVILEGED REGISTERS

The internal processor register (IPR) space provides access to many types of CPU control and status registers such as the memory-management base registers, parts of 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.

Table 8-1 summarizes architecturally required IPRs, and Table 8-2 lists all IPRs. Those internal processor registers that require further explanation are described below. Reference to general registers means R0 through R13, the SP, and the PC (see Chapter 1, Basic Architecture). Registers referenced by the MTPR and MFPR instructions are designated processor registers and appear in the processor register space.

8.1 PER-PROCESS REGISTERS AND CONTEXT SWITCHING

Several per-process registers are loaded from the PCB during a context load operation and, with the exception of the memory management registers, address space number register, PME, and AST level, are written back to the PCB during a context save operation (see Chapter 6). 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.

An implementation may retain some or all per-process stack pointers only in the PCB. In this case, MTPR and MFPR for these registers must access the corresponding PCB locations. However, implementations that have per-process stack pointers in hardware scratchpads are not required to access the corresponding PCB locations for MTPR and MFPR. The PCB locations get updated when a SVPCTX instruction is executed.

Implementations are not allowed to retain copies of the per-process memory-management registers (P0BR, P0LR, P1BR, P1LR) only in the PCB. The processor must implement each of these registers in hardware. The same applies to the address space number register, ASN, if supported.

It is possible that some implementations will retain either ASTLVL or PME or both only in the PCB. These processors will implement MTPR and MFPR for those registers as a no-op, at least in the sense that the destination operand (for MFPR) or IPR (for MTPR) is not written. Other implementations may copy either or both of these registers from the PCB into scratchpad registers.

The SVPCTX instruction does not write the per-process memory-management registers, ASN, ASTLVL, and PME back into the PCB. To ensure that the PCB is always correctly updated, software must use the following convention when referencing any of these registers.

1. Write. Software must first write the value directly into the proper location in the current PCB by using a MOVL (for example), then execute an MTPR with the same source as the MOVL. Implementations that do not retain internal copies of these registers will effectively no-op the MTPR instruction. They will not take a reserved operand fault which would normally occur for a non-existent register.

2. Read. Software can read the value directly from the proper location in the current PCB by using a EXTZV (for example). It is not necessary to execute an MFPR from the corresponding internal register, since the PCB location always contains an updated value due to the software convention for writing these registers.

8.2 STACK-POINTER IMAGES

Reference to SP (the stack pointer) in the general registers will access one of five possible stack pointers (user, supervisor, executive, kernel, or interrupt) depending on the values of PSL<CUR_MOD> and PSL<IS> (see Chapter 5). Additionally, software can access any of the five stack pointers (including the one currently selected by PSL<CUR_MOD> and PSL<IS>) with 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 internal processor register address space by an MTPR or MFPR instruction. This means that an MFPR or MTPR to the KSP (if PSL<IS>=0) or the ISP (if PSL<IS>=1) is equivalent to a MOVL from or to the SP.

8.3 MTPR AND MFPR INSTRUCTIONS

MTPR   Move To Processor Register

Format:

```
opcode  src.rl, procreg.rl
```

Operation:

```
if PSL<VM> EQLU 1 AND VMPSL<CUR_MOD> EQLU 0 then
        {initiate VM-emulation trap};
        ! Src and procreg are pushed in exception frame.

if PSL <CUR_MOD> NEQ 0 then {privileged-instruction
        fault};
IPR[procreg] <- src;
```

Condition Codes:

```
N <- UNPREDICTABLE
Z <- UNPREDICTABLE
V <- UNPREDICTABLE
C <- UNPREDICTABLE
```

Exception:

privileged instruction
VM emulation

Opcode:

```
DA    MTPR    Move To Processor Register
```

Description:

If the processor is executing a virtual machine and the virtual machine is in kernel mode, then a VM-emulation trap is initiated. Otherwise, MTPR loads the source operand specified by source into all copies of the processor register specified by procreg that are implemented on the scalar and vector processors. The procreg operand is a longword that contains the processor register number. Execution may have register-dependent side effects.

Notes:

1. A privileged instruction fault occurs if instruction execution is attempted in other than kernel mode (PSL<CUR_MOD> EQLU 0) or VM-kernel mode (PSL<VM> EQLU 1 and VMPSL<CUR_MOD> EQLU 0).

2. If a register is implemented only as a PCB location, MTPR to that register has no effect.

3. In kernel mode, the operation of the processor is UNDEFINED after execution of MTPR to a read-only register, MTPR to a nonexistent register, MTPR of a non-zero value to an MBZ field, or MTPR of a reserved value to a register. The preferred implementation is to cause a reserved-operand fault. In VM-kernel mode, a reserved-operand fault will not occur because the processor does not check accessibility or existence of the target register, nor does the processor check for MBZ or reserved values. These checks should be made in software.

4. After an MTPR instruction, the condition codes are UNPREDICTABLE, unless noted otherwise under the description of the specific processor register.

MFPR   Move From Processor Register

Format:

```
        opcode procreg.rl, dst.wl
```

Operation:

```
        if PSL<VM> EQLU 1 AND VMPSL<CUR_MOD> EQLU 0 then
                {initiate VM-emulation trap};
                ! Procreg and address of dst are pushed in
                ! exception frame.

        if PSL <CUR_MOD> NEQ 0 then {privileged-instruction
                fault};
        dst <- IPR[procreg];
```

Condition Codes:

```
        N <- UNPREDICTABLE
        Z <- UNPREDICTABLE
        V <- UNPREDICTABLE
        C <- UNPREDICTABLE
```

Exception:

privileged instruction
VM emulation

Opcode:

DB	MFPR	Move From Processor Register

Description:

If the processor is executing a virtual machine and the virtual machine is in kernel mode, then a VM-emulation trap is initiated. Otherwise, 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-dependent side effects.

Notes:

1. A privileged instruction fault occurs if instruction execution is attempted in other than kernel mode or VM-kernel mode (that is, PSL<VM> EQLU 1 and VMPSL<CUR_MOD> EQLU 0).

2. If a register is implemented only as a PCB location, MFPR from that register has no effect.

3. In kernel mode, the operation of the processor is UNDEFINED after execution of MFPR from a register that does not exist, or after execution of MFPR from a write-only register. The preferred implementation is to cause a reserved-operand fault. In VM-kernel mode, a reserved-operand fault will not occur because the processor does not check existence or accessibility of the source register. These checks should be made in software.

4. After an MFPR instruction, the condition codes are UNPREDICTABLE, unless noted otherwise under the description of the specific processor register.

Table 8-1: Required Internal Processor Registers

Name	Mnemonic	Decimal	Hex	Type	Scope
kernel stack pointer	KSP	0	0	R/W	process
executive stack pointer	ESP	1	1	R/W	process
supervisor stack pointer	SSP	2	2	R/W	process
user stack pointer	USP	3	3	R/W	process
interrupt stack pointer	ISP	4	4	R/W	CPU
address space number*	ASN	6	6	R/W	process
P0 base register+	P0BR	8	8	R/W	process*
P0 length register+	P0LR	9	9	R/W	process*
P1 base register+	P1BR	10	A	R/W	process*
P1 length register+	P1LR	11	B	R/W	process*
system base register+	SBR	12	C	R/W	CPU/VP
system length register+	SLR	13	D	R/W	CPU/VP
CPU identification*	CPUID	14	E	R	CPU
process control block base	PCBB	16	10	R/W	CPU
system control block base	SCBB	17	11	R/W	CPU
interrupt priority level	IPL	18	12	R/W	CPU
AST level	ASTLVL	19	13	R/W	process
software interrupt request	SIRR	20	14	W	CPU
software interrupt summary	SISR	21	15	R/W	CPU
interval clock control*	ICCS	24	18	R/W	CPU
next interval count*	NICR	25	19	W	CPU
interval count*	ICR	26	1A	R	CPU
time of year*	TODR	27	1B	R/W	CPU
console receiver status*	RXCS	32	20	R/W	CPU
console receiver data buffer*	RXDB	33	21	R	CPU
console transmit status*	TXCS	34	22	R/W	CPU
console transmit data buffer*	TXDB	35	23	W	CPU
memory management enable	MAPEN	56	38	R/W	CPU
translation buffer invalidate all	TBIA	57	39	W	CPU*
translation buffer invalidate single	TBIS	58	3A	W	CPU*
translation buffer invalidate ASN*	TBIASN	59	3B	W	CPU*
translation buffer invalidate system*	TBISYS	60	3C	W	CPU*
performance monitor enable*	PME	61	3D	R/W	process
system identification	SID	62	3E	R	CPU
translation buffer check	TBCHK	63	3F	W	CPU
virtual-machine PSL	VMPSL	102	66	R/W	VM
vector processor status register+	VPSR	144	90	R/W	VP
vector arithmetic exception register+	VAER	145	91	R	VP
vector memory activity check+	VMAC	146	92	R	VP
vector TB invalidate all+	VTBIA	147	93	W	VP
vector state address register+	VSAR	148	94	R/W	VP

* Implementations are not required to include CPUID, ASN, NICR, ICR, TODR, RXCS, RXDB, TXCS, TXDB, TBIASN, TBISYS, PME, or VMPSL. Only a subset of ICCS is required. For more detail, refer to Chapter 11, Implementation Options.

+ Implementations which do not support a vector processor are not required to include VPSR, VAER, VMAC, VTBIA, or VSAR. If an implementation supports an optional vector processor but it is not installed, reading VPSR and VMAC returns zero and writing VPSR has no effect. A vector processor is not required to include P0BR, P0LR, P1BR, P1LR, SBR, SLR, VSAR. If the vector processor does not have a private translation buffer, then it does not have a copy of VTBIA; however, in that case writing VTBIA has no effect. For more details see 13.

Key: process - one copy per process, residing on the scalar processor, loaded by LDPCTX

process* - one copy per process on the scalar processor and optionally one more copy on the vector processor, loaded into the scalar processor by LDPCTX

CPU - one copy per scalar processor, not affected by LDPCTX

CPU* - one copy per scalar/vector pair, not affected by LDPCTX

VM - one copy per virtual machine, not affected by LDPCTX

VP - one copy per vector processor, not affected by LDPCTX

CPU/VP - one copy per scalar processor and optionally one more copy on the vector processor, not affected by LDPCTX

R - register can be read but cannot be written

W - register can be written but cannot be read

R/W - register can be both read and written

Table 8-2: All Internal Processor Registers

Number		Name	7	7	7	u	8	8	8	u	C	6	V	9	R	N	M	
			3	5	8	V	2	6	8	V	V	4	V	0	V	V	A	
			0	0	0	A	0	0	0	A	A	0	A	0	A	A	R	
						X	0	0	0	X	X	0	X	0	X	X	I	
						1				2							A	
																	H	
0	0	KSP	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
1	1	ESP	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
2	2	SSP	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
3	3	USP	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
4	4	ISP	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
5	5																	
6	6	ASN																
7	7																	
8	8	P0BR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
9	9	P0LR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
10	A	P1BR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
11	B	P1LR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
12	C	SBR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
13	D	SLR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
14	E	CPUID												Y	Y			
15	F	WHAMI												Y				
16	10	PCBB	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
17	11	SCBB	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
18	12	IPL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
19	13	ASTLVL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
20	14	SIRR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
21	15	SISR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
22	16	IPIR					Y											
23	17	CMIERR		Y														
24	18	ICCS	Y	Y	Y	s	Y	Y	Y	s	s	s	s	Y	Y	i	s	
25	19	NICR	Y	Y	Y		Y	Y	Y					Y	Y	i		
26	1A	ICR	Y	Y	Y		Y	Y	Y					Y	Y	i		
27	1B	TODR	Y	Y	Y		Y	Y				Y		Y	Y	Y	Y	
28	1C	CSRS	Y	Y								n			Y	Y	n	
29	1D	CSRD	Y	Y								n			Y	Y	n	
30	1E	CSTS	Y	Y								n			Y	Y	n	
31	1F	CSTD	Y	Y								n			Y	Y	n	

6400 = 6000-400

s = subset implementation

n = not supported or fully implemented

i = whether the register is implemented and whether the implementation is full or subset depends on the system implementation the processor is in.

Table 8-2: All Internal Processor Registers (continued)

Number		Name	730	750	780	uVAX1	8200	8600	8800	uVAX2	CVAX	6400	VVAX	9000	RVAX	NVAX	MARIAH	
32	20	RXCS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
33	21	RXDB	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
34	22	TXCS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
35	23	TXDB	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
36	24	TBDR	Y	Y			Y											
37	25	CADR	Y	Y		Y	Y											
38	26	MCESR	Y	Y		Y	Y		Y				Y			Y	Y	
39	27	CAER	Y	Y														
40	28	ACCS	Y	Y	Y		Y	Y		Y		Y	Y		Y		Y	
41	29	SAVGPR			*					+					+			(*ACCR) (+SAVISP)
42	2A	SAVPC				Y				Y		Y			Y	Y	Y	
43	2B	SAVPSL				Y						Y			Y	Y	Y	
44	2C	WCSA			Y		Y								*			(* MCESR)
45	2D	WCSD			Y		Y								*			(* BCWBLK)
46	2E	WCSCAM					Y								*			(* PCIBLK)
47	2F	TBTAG										Y					Y	
48	30	SBIFS	Y		Y													
49	31	SBIS	Y		Y													
50	32	SBISC	Y		Y													
51	33	SBIMT	Y		Y													
52	34	SBIER	Y		Y													
53	35	SBITA	Y		Y													
54	36	SBIQC	Y		Y													
55	37	IORESET	Y	Y		Y				Y		Y	Y			Y	Y	
56	38	MAPEN	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
57	39	TBIA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
58	3A	TBIS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
59	3B	TBDATA		Y								Y					Y	
59	3B	TBIASN																
60	3C	MBRK			Y													
60	3C	TBISYS																
61	3D	PME	Y	Y	Y		Y	Y	Y				Y	Y				
62	3E	SID	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)
63	3F	TBCHK	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	(required)

6400 = 6000-400

Table 8-2: All Internal Processor Registers (continued)

Number		Name	730	750	780	uVAX1	8200	8600	8800	uVAX2	CVAX	6400	VVAX	9000	RVAX	NVAX	MARIAH
64	40	PAMACC						Y								-	
65	41	PAMLOC						Y								-	
66	42	CSWP						Y						Y		-	
67	43	MDECC						Y								-	
68	44	MENA						Y								-	
69	45	MDCTL						Y									
70	46	MCCTL						Y									
71	47	MERG						Y									
72	48	CRBT						Y						Y			
73	49	DFI						Y									
74	4A	EHSR						Y									
75	4B																
76	4C	STXCS						Y									
77	4D	STXDB						Y									
78	4E	ESPA						Y									
79	4F	ESPD						Y									
80	50	RXCS1					Y										
81	51	RXDB1					Y										
82	52	TXCS1					Y										
83	53	TXDB1					Y										
84	54	RXCS2					Y										
85	55	RXDB2					Y										
86	56	TXCS2					Y										
87	57	TXDB2					Y										
88	58	RXCS3					Y										
89	59	RXDB3					Y										
90	5A	TXCS3					Y										
91	5B	TXDB3					Y										
92	5C	RXCD					Y										
93	5D	CACHEX					Y										
94	5E	BINID					Y										
95	5F	BISTOP					Y										

6400 = 6000-400

- = address of IPR decoded by processor, but register is either not meant for normal system use or is not usable at all

Table 8-2: All Internal Processor Registers (continued)

Number		Name	730	750	780	uVAX1	8200	8600	8800	uVAX2	CVAX	6400	VVAX	9000	RVAX	NVAX	MARIAH	
96	60																	
97	61																	
98	62																	
99	63																	
100	64	MEMSIZE											Y					
101	65	KCALL											Y					
102	66	VMPSL							m							m		
103	67																	
104	68																	
105	69																	
106	6A	CPUCNF												Y				
107	6B	ICIR												Y				
108	6C	RXFCT												Y				
109	6D	RXPRM												Y				
110	6E	TXFCT												Y				
111	6F	TXPRM												Y				
112	70	BCIDX															Y	
113	71	BCBTS										Y					*	(* BCSTS)
114	72	BCP1TS										Y					*	(* BCCTL)
115	73	BCP2TS										Y					*	(* BCERA)
116	74	BCRFR										Y					*	(* BCBTS)
117	75	BCIDX										Y					*	(* BCDET)
118	76	BCSTS										Y					*	(* BCERT)
119	77	BCCTL										Y					-	
120	78	BCERR										Y					-	
121	79	BCFBTS										Y					-	
122	7A	BCFPTS										Y					-	
123	7B	VINTSR										Y					Y	
124	7C	PCTAG										Y				*	Y	(* JTAGCR)
125	7D	PCIDX										Y				*	Y	(* ECR)
126	7E	PCERR										Y				-	Y	
127	7F	PCSTS										Y				-	Y	

6400 = 6000-400
m = Supported in the VM-variant only

- = address of IPR decoded by processor, but register is either not meant for normal system use or is not usable at all

Table 8-2: All Internal Processor Registers (continued)

Number		Name	730	750	780	uVAX1	8200	8600	8800	uVAX2	CVAX	6400	VVAX	9000	RVAX	NVAX	MARIAH	
128	80	NMION							Y							-		
129	81	INOP							Y							-		
130	82	NMIFSR							Y							-		
131	83	NMISILO							Y							-		
132	84	NMIEAR							Y							-		
133	85	CCR							Y							-		
134	86	REVR1							Y							-		
135	87	REVR2							Y							*		(* VPAMODE)
136	88	CLRTOSTS							Y							-		
137	89															-		
138	8A															-		
139	8B															-		
140	8C															-		
141	8D															-		
142	8E															-		
143	8F															-		
144	90	VPSR										v	w	v	v	v	v	
145	91	VAER										v	w	v	v	v	v	
146	92	VMAC										v	w	v	v	v	v	
147	93	VTBIA										v	w	v	v	v	v	
148	94	VSAR																
149	95	Reserved																
150	96	Reserved																
151	97	Reserved																
152	98	Reserved																
153	99	Reserved																
154	9A	Reserved																
155	9B	Reserved																
156	9C	Reserved																
157	9D	VIADR										v					v	
158	9E	VIDLO										v					v	
159	9F	VIDHI										v					v	

6400 = 6000-400

v = implemented when optional vector processor is present

w = VVAX is architecturally capable of supporting the register but the current VVAX implementation does not

Table 8-2: All Internal Processor Registers (continued)

Number	Name	730	750	780	uVAX1	8200	8600	8800	uVAX2	CVAX	6400	VVAX	9000	RVAX	NVAX	MARIAH
160 A0	CCTL														Y	
161 A1																
162 A2	BCDECC														Y	
163 A3	BCETSTS														Y	
164 A4	BCETIDX														Y	
165 A5	BCETAG														Y	
166 A6	BCEDSTS														Y	
167 A7	BCEDIDX														Y	
168 A8	BCEDHI														Y	
169 A9	BCEDLO														Y	
170 AA	BCEDECC														Y	
171 AB	CEFADR														Y	
172 AC	CEFSTS														Y	
173 AD																
174 AE																
175 AF																
176 B0	NESTS														Y	
177 B1	NEOADR														Y	
178 B2	NEOCMD														Y	
179 B3	NEDATHI														Y	
180 B4	NEDATLO														Y	
181 B5	NEICMD														Y	
182 B6																
183 B7																
184 B8	VADR														Y	
185 B9	VCMD														Y	
186 BA	VQWLO														Y	
187 BB	VQWHI														Y	
188 BC																
189 BD																
190 BE																
191 BF																

6400 = 6000-400

Number	Name	730	750	780	uVAX1	8200	8600	8800	uVAX2	CVAX	6400	VVAX	9000	RVAX	NVAX	MARIAH
192 C0																
193 C1																
194 C2																
195 C3																
196 C4																
197 C5																
198 C6																
199 C7																
200 C8																
201 C9																
202 CA																
203 CB																
204 CC																
205 CD																
206 CE																
207 CF																
208 D0	VMAR														Y	
209 D1	VTAG														Y	
210 D2	VDATA														Y	
211 D3	ICSR														Y	
212 D4															-	
213 D5																
214 D6															-	
215 D7															-	
216 D8																
217 D9																
218 DA																
219 DB																
220 DC																
221 DD																
222 DE																
223 DF																

6400 = 6000-400

- = address of IPR decoded by processor, but register is either not meant for normal system use or is not usable at all

Number	Name	7	7	7	u	8	8	8	u	C	6	V	9	R	N	M
		3	5	8	V	2	6	8	V	V	4	V	0	V	V	A
		0	0	0	A	0	0	0	A	A	0	A	0	A	A	R
					X	0	0	0	X	X	0	X	0	X	X	I
					1				2							A
																H
224 E0															-	
225 E1															-	
226 E2															-	
227 E3															-	
228 E4															-	
229 E5															-	
230 E6															-	
231 E7	PAMODE														Y	
232 E8	MMEADR														Y	
233 E9	MMEPTE														Y	
234 EA	MMESTS														Y	
235 EB																
236 EC	TBADR														Y	
237 ED	TBSTS														Y	
238 EE																
239 EF																
240 F0	PCADR														Y	
241 F1	PCSTS														Y	
242 F2	PCCTL														Y	
243 F3																
244 F4																
245 F5																
246 F6																
247 F7																
248 F8																
249 F9																
250 FA																
251 FB																
252 FC																
253 FD																
254 FE																
255 FF																

0000 0100 - 00FF FFFF	Are not implemented by any processor
0100 0000 - FFFF FFFF	Are used by NVAX to access Bcache and Pcache registers

6400 = 6000-400

- = address of IPR decoded by processor, but register is either not meant for normal system use or is not usable at all

8.3.1 CPU Identification Register

In a multiprocessor system, software occasionally must know on what processor it is running. The CPU identification register (CPUID) exists for this purpose. CPUID contains a different value in each processor of a multiprocessor system, and is in the range 0..K-1 in a K-processor system. On processors that implement the ASN register, LDPCTX and SVPCTX make use of the CPUID register and the PRVCPU field in the PCB to determine when a process moves from one processor to another. The CPUID value 255 is reserved for use by software, to ensure that there is a CPUID value that is never matched. Figure 8-1 shows the CPUID register. _\The value in CPUID may be built into the hardware, written by the processor or the console during system initialization, or written by software during operating-system initialization, depending on processor implementation._\

```
 3
 1                                             8 7             0
+-----------------------------------------------+---------------+
|                       MBZ                     | CPU number    |
+-----------------------------------------------+---------------+
```

Figure 8-1 CPU Identification Register (CPUID)

8.3.2 System Identification Register

The system identification register (SID) specifies the processor type and includes an implementation-dependent field. The processor type field is used by software in handling implementation-dependent processor features. The implementation-dependent field typically specifies additional information, such as hardware revision level and microcode revision level, and is included in the error log to more finely distinguish processor types. The SID is shown in Figure 8-2. Table 8-3 shows the processor type codes. See Appendix B for details of particular implementations.

8.3.3 System-type Register

The system-type register (SYS_TYPE) augments the SID, which identifies processors. SYS_TYPE allows software to distinguish between different systems based on the same processor. It is implemented on systems that provide console functions by means of VAX instructions in I/O space. Systems that include separate console processors do not include SYS_TYPE, but may provide information for the same purpose through console functions. The SYS_TYPE<31:24> subfield specifies the system type and distinguishes between different systems based on the same processor, all of which would have the same SID. Table 8-3 shows the system type codes assigned for each SID value.

SYS_TYPE is located at the beginning of I/O space plus 40004 (hex). It normally resides in a ROM and is included in the ROM's checksum. Figure 8-3 shows the SYS_TYPE register and its four subfields.

- The SYS_TYPE<31:24> subfield is assigned by the VAX Architecture Group.

- The REV_LEVEL subfield, SYS_TYPE<23:16>, specifies the hardware revision level of the system (not the processor) and is changed whenever an improvement is made to the system hardware. The revision level is always one or greater. It is assigned by the hardware design groups and is not published in this standard.

- The SYS_DEPENDENT subfield, SYS_TYPE<15:8>, specifies minor variations (within a common SYS_TYPE) that require few or no software changes, or that might be licensed differently. Examples might be systems with different environmental requirements or differences in performance or packaging. The variation number is assigned by the hardware design groups and is not published in this standard.

- The LICENSE_ID subfield, SYS_TYPE<7:0>, specifies what type of software license is required. One and only one of the bits in the field may be set. Table 8-4 shows the permissible values and their meanings. They are assigned by the VAX Architecture Group. \ At the present time, Jay Nichols (KRYPTN::NICHOLS) and Kathy Morse (STAR::MORSE) are making these assignments. \

Table 8-3: SID<31:24> and SYS_TYPE<31:24> Assignments

SID	Processor	SYS_TYPE	System
0	Reserved to DIGITAL		
1	VAX-11/780; VAX-11/782; VAX-11/785		
2	VAX-11/750		
3	VAX-11/730		
4	VAX 8600; VAX 8650		
5	VAX 8200, 8300; VAX 8250, 8350; VS8000		
6	VAX 8700,8800,8810,8820-N; VAX 8500,8530,8550		
7	MicroVAX-I		
8	MicroVAX-II chip		
		0	Reserved to Digital
		1	MicroVAX II
		2-3	Reserved to DIGITAL
		4	MicroVAX 2000 VAXstation 2000 PC/LAN SERVER 2000 Enterprise (DEC/Allen Bradley) MicroVAX Information Processor Module (aka Industrial <VAXserver, MicroVAX> 2000)
		5	VAXterm
		6	VAX 9000 Console
		7-127	Reserved to DIGITAL
		128-255	Reserved to system owners
9	Reserved to DIGITAL		
10	CVAX chip		
		0	Reserved to DIGITAL
		1	MicroVAX 3500 or 3600 VAXserver 3500 or 3600 VAXstation 3200 or 3500 MicroVAX 3300, 3400 VAXserver 3300, 3400 MicroVAX 3800, 3900 VAXserver 3800, 3900
		2	VAX 62n0 VAX Fileserver 62n0 VAX 63n0 VAX Fileserver 63n0 Raytheon Model 860
		3	VAXstation 3520, 3540

Table 8-3 SID<31:24> and SYS_TYPE<31:24> Assignments (cont)

SID = Processor	SYS_TYPE	System
10 = CVAX chip		
	4	VAXstation 3100
	5	Reserved to DIGITAL
	6	Reserved to DIGITAL
	7	VAX 5200FT
	8-127	Reserved to DIGITAL
	128-255	Reserved to CSS and customers
11 = REX520 chip		
	0-1	Reserved to DIGITAL
	2	VAX 6000-4n0
	3-255	Reserved to DIGITAL
12 = Reserved to DIGITAL		
13 = Reserved to DIGITAL		
14 = VAX 9000		
15 = Reserved to DIGITAL		
16 = rt/uVAX (chip 78R32)		
	0	Reserved to DIGITAL
	1	rtVAX 1000
	2-127	Reserved to DIGITAL
	128-255	Reserved to CSS and customers
17 = VAX 8820, 8830, 8840		
18-128 = Reserved to Digital		
129-255 = Used for Digital MIPS processors		

NOTE

\ A copy of the complete SID list can be obtained from the VAX Architecture group on a need-to-know basis. Send mail to EAGLE1::SRM for further details.\

Table 8-4: Bit Assignments in the LICENSE_ID Subfield

Extent	Permissible Values
<0>	1 if a Timesharing system; 0 otherwise (m_u)
<1>	1 if a Single User system; 0 otherwise (s_u)
<7:2>	Reserved to Digital and must be zero until Digital defines them.

Note: The field must contain only a single one bit.

```
 31              24 23                                           0
+-----------------+----------------------------------------------+
|       TYPE      |                type dependent                |
+-----------------+----------------------------------------------+
```

Figure 8-2 System Identification Register (SID)

```
 3              2 2             1 1             8 7              0
 1              4 3             6 5
+---------------+---------------+---------------+----------------+
|   SYS_TYPE    |   REV_LEVEL   | SYS_DEPENDENT |   LICENSE_ID   |
+---------------+---------------+---------------+----------------+
```

Figure 8-3 System-Type Register (SYS_TYPE)

8.3.4 Time-of-Year Clock Register

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 time-of-year clock is required by VMS for power fail recovery, the swapper, and the device drivers for the RL02, RX02, and the TS11. Also, including the optional battery back-up is the strongly preferred implementation. Unattended operation and some communication protocols depend on the system maintaining the correct time over power failures.\

The time-of-year clock consists of one longword register, shown in Figure 8-4. The register forms an unsigned 32-bit binary counter that is driven by a precision clock source with at least .0025 percent 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 (say, a month), it can check for loss of time after a power restore by checking the clock value.

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 preferred implementation.\

8.3.5 Interval Clock Registers

The interval clock is used for accounting, for time-dependent events, and to maintain the software date and time. It provides an interrupt at IPL 22 or 24 at programmed intervals. The preferred implementation is at IPL 22. The counter is incremented at 1-microsecond intervals, with at least .01 percent accuracy (8.64 seconds per day). The clock interface consists of three internal processor registers, illustrated in Figure 8-5 and Table 8-5:

- Interval Count Register (ICR) -- The interval count register is a read-only register incremented once every microsecond. Upon a carry out (overflow) from bit <31>, it is automatically loaded from NICR; an interrupt is generated if the interrupt is enabled. That is, the value of ICR on successive microseconds will be FFFFFFFD (hex), FFFFFFFE, FFFFFFFF, <value of NICR>.

- Next Interval Count Register (NICR) -- This reload register is a write-only register that holds the value to be loaded into ICR when ICR overflows. The value is retained when ICR is loaded.

- Interval Clock Control Status Register (ICCS) -- The ICCS register contains control and status information for the interval clock.

Thus, to use the interval clock, load the negative of the desired interval into NICR. Then an MTPR #^X51,#PR$_ICCS will enable interrupts, reload ICR with the NICR interval, and set run. Every "interval count" microseconds will cause interrupt to be set and an interrupt to be requested. The interrupt routine should execute an MTPR #^XC1,#PR$_ICCS to clear the interrupt. If interrupt has not been cleared (the interrupt has not been handled) by the time of the next ICR overflow, error will be set. Note: If NICR is written while the clock is running, the clock may lose or add a few ticks. If the interval clock interrupt is enabled, this may cause the loss of an interrupt.

Processor initialization leaves ICR and NICR UNPREDICTABLE, clears ICCS <6> and <0>, and leaves the rest of ICCS UNPREDICTABLE.

NOTE

> Processors may omit NICR and ICR, and are required only to implement ICCS<IE>. If this bit is set, an interrupt request at IPL 22 is generated once every 10 milliseconds.

```
 31                                                          0
+------------------------------------------------------------+
|                 time of year since setting                 |
+------------------------------------------------------------+
```

Figure 8-4 Time of Year (TODR)

```
 31                                                          0
+------------------------------------------------------------+
|                       interval count                       |
+------------------------------------------------------------+
```

a. Interval Count (ICR)

```
 31                                                          0
+------------------------------------------------------------+
|                     next interval count                    |
+------------------------------------------------------------+
```

b. Next Interval Count (NICR)

```
 31 30                                           8 7 6 5 4 3   1 0
+--+---------------------------------------------+-+-+-+-+-----+-+
|  |                     MBZ                     | | | | | MBZ | |
+--+---------------------------------------------+-+-+-+-+-----+-+
```

c. Interval-Clock Control and Status (ICCS)

```
                                                   6
+-------------------------------------------------+-+-----------+
|                       MBZ                       | |    MBZ    |
+-------------------------------------------------+-+-----------+
```

d. Subset Interval-Clock Control and Status (ICCS)

Figure 8-5 Clock Registers

Table 8-5: Fields of the ICCS Register

Name	Extent	Description
error	<31>	When ICR overflows, if interrupt is already set, then error is set. Thus, error indicates a missed clock tick. Write 1 to clear. Note: On a subset implementation, this bit must be zero.
interrupt	<7>	Set by hardware every time ICR overflows. If interrupt-enable is set, then an interrupt is also generated. Writing a 1 to this bit with MTPR clears it, thereby reenabling the clock tick interrupt. Note: On a subset implementation, this bit must be zero.
interrupt enable	<6>	When set, an interrupt request is generated every time ICR overflows (every time interrupt is set). When clear, no interrupt is requested. Similarly, if interrupt is already set and the software sets interrupt enable, an interrupt is generated. That is, an interrupt is generated whenever the function {interrupt enable AND interrupt} changes from 0 to 1. Processor initialization clears interrupt enable.
single step	<5>	If run is clear, each time this bit is set, ICR is incremented by one. Write only. Note: On a subset implementation this bit must be zero.
transfer	<4>	When a 1 is written to this bit, NICR is transferred to ICR. Write only. Note: On a subset implementation this bit must be zero.
run	<0>	When set, ICR increments each microsecond. When clear, ICR does not increment automatically. Processor initialization clears run. Note: On a subset implementation this bit must be zero.

Change History:

Revision J. Rich Brunner, Tim Leonard, December 1989

- o MTPR updates copies of IPRs on vector processor.
- o Added Vector IPRs, and RVAX, NVAX, Mariah, and Aquarius IPRs. Re-worked required IPR list to expand scope for vectors. Made corrections where necessary.
- o removed SPTEP IPR.
- o Added ASN IPR and TBIASN. Revised Process ID text per ECO 119
- o Process Memory Management Registers must not reside only in PCB. Processor must have hardware copy.
- o Updated SID and SYS_TYPE codes for announced products. Included combined table of SID and SYSTYPES.
- o Add ECO 103 -- Process IDs.
- o Added note from ECO 90 that condition codes are UNPREDICTABLE after MxPR, EXCEPT where noted under the IPR description.

Revision H. Tim Leonard, May 1987.

- o Add the SPTEP register.
- o Support virtual machines.

Revision F. Al Thomas, November, 1986.

- o Condition codes are UNPREDICTABLE after MxPR.
- o Remove references to subset processors.

Revision E. Tim Leonard, September 1986.

- o Clarify fields for subset implementation of ICCS.

Revision D. Tim Leonard, March 1985.

- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Make clear that ICR is reloaded from NICR at once, rather than holding the value 0 for one tick.
- o Move console terminal register descriptions to Chapter 11.
- o Writing to a read only IPR, reading from a write-only IPR, referencing a nonexistent IPR, writing a reserved value to an IPR, all UNDEFINED behavior.
- o Writing to NICR while the clock is running may cause the clock to gain or lose a few ticks.
- o Change INSV to MOVL in the section talking about IPRs implemented only in the PCB. INSV doesn't set the condition codes right.
- o Change timer IPL to 22 for future processors.
- o Subset timer for MicroVAX.

Revision 6, add CPU dependent registers. Dileep Bhandarkar, 26 July 1982.

- o Add processor type code for VAX-11/750, VAX-11/730.
- o List implementation-specific IPRs for 780, 750, and 730.
- o Move Console and Bootstrapping to separate chapter.
- o Fault on MTPR to read-only and MFPR from write-only IPRs.
- o Change MTPR and MFPR to per-process IPRs in PCB only.
- o Reserve processor type codes for VENUS, SCORPIO, NAUTILUS.

Revision 5. Dileep Bhandarkar and Tom Eggers, 26 July 1980.

- 11/780 Dependent SID format.
- Clarify stack pointer references.
- TOY clock required.

Revision 4. Peter Conklin and Ted Taylor, 4 August 1978.

- Remove I/O section to chapter 8.
- MTPR and MFPR of xSP always works (stack register ECO).
- Move IPL register to chapter 6.
- Add SID definition.
- Add console terminal definition.
- Add clock definition.
- Move TBIA and TBIS to chapter 5.
- Move VAX-11/780 error registers to chapter 11.
- Document VAX-11/780 accelerator registers.
- Document VAX-11/780 control store registers.
- Fault on MTPR to read-only register or MFPR from write-only register.
- Add register numbers.
- Add VAX-11/780 registers to table.
- IPL is R/W not W.
- ICR is read-only, NICR is write-only; not R/W.
- Add ASTLVL in 7; MAPEN in 5; PME in 7; TODR in 9.
- Add initial values of all registers, TB, cache.
- Add boot set of general registers.
- Make console terminal mandatory.
- SID<23:0> now type dependent.
- Add RO, RW, WC, WO field notations.
- Clarify identity of function for machines having the same register number.
- Console halt only between instruction; not an interrupt.
- Change reset to initialize.
- Change instruction-buffer clear from deposit to continue.
- Console virtual relative to PSL<CUR_MOD>.
- Add system restart section.
- Move xSP to chapter 6.
- Move process registers to chapter 7.
- Clarify when RXCS and TXCS cause interrupts.
- Make ACCS type be 8 bits.
- Note diagnostics use NOP for scope synchronization.
- Deposit/virtual command maintains PTE.
- Add 780 SID spec.
- Name change CRxx, CTxx to RXxx, TX.
- Missing TODR is always 0.

Revision 3, added console specs. Learoyd, 3 June 1976.

- Added notion of UNIBUS space.
- Added I/O register constraint rules.
- Added I/O space interlocking.
- Added MTPR, MFPR descriptions.
- MTPR, MFPR require kernel mode.
- Added clocks, bootstrap (no spec yet).
- Added WCS register (no spec yet)
- Added FCS Internal Registers.
- Added console function section.

- o Deleted description of multiple register sets.
- o Make WCS registers implementation-dependent.
- o Delete PSL in IPR space.
- o Add discussion of per-process registers in privileged register space.
- o Added description of interval timer.
- o Add description of time-of-day clock.

Revision 2. Dave Rodgers, 21 February 1976.

- o Added list of privileged registers.
- o Expanded I/O description.
- o Expanded privileged register description.

Revision 1. Dave Rodgers, 10 October 1975.

CHAPTER 9

PDP-11 COMPATIBILITY MODE

Implementation of PDP-11 compatibility mode is optional. Processors that do implement compatibility mode do so as specified in this chapter. Operating system software may emulate compatibility mode on processors that omit this mode.

VAX compatibility-mode hardware, in conjunction with a compatibility-mode software executive (which runs in VAX native mode), can emulate the environment provided to user programs on a PDP-11 system. This environment does not include the following features of normal PDP-11 system operation:

- o Privileged instructions such as HALT and RESET
- o Special instructions such as traps and WAIT
- o Access to internal processor registers such as the PSW and the console switch register
- o Direct access to trap and interrupt vectors
- o Direct access to I/O devices
- o Interrupt servicing
- o Stack overflow protection
- o Alternate general register sets
- o Any processor mode other than user (that is, kernel and supervisor modes are not supported) and separate I and D spaces
- o 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.

9.1 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 autoincrement modes in JMP and JSR do not affect the new PC. Side effects caused by a source address calculation, however, 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.

The operands of some PDP-11 instructions are implied by the instruction type, whereas others are specified as part of the instruction. The different kinds of operand specifiers appearing in PDP-11 instructions are shown in Figure 9-1. Address mode operand specifiers include a 3-bit mode field, specifying one of eight modes: register, register deferred, autoincrement, autoincrement deferred, autodecrement, autodecrement deferred, index, or index deferred mode. These modes are discussed in the following sections.

9.1.1 Register Mode

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, that is, 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.

9.1.2 Register Deferred Mode

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.

9.1.3 Autoincrement 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); the register is then replaced by the result. If Rn denotes SP or PC, the register is incremented by 2 and the register is replaced by the result.

```
OA = Rn
if n LEQ 5 then Rn <- Rn + size else Rn <- Rn + 2
operand = (OA)
```

If Rn denotes PC, immediate data follows the instruction. The mode is termed immediate mode.

The assembler notation for autoincrement mode is (Rn)+. For immediate mode, the notation is #constant where constant is the data immediately following the instruction.

9.1.4 Autoincrement Deferred 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)
```

If Rn denotes PC, a 16-bit address follows the instruction. The mode is termed absolute mode.

The assembler notation for autoincrement deferred mode is @(Rn)+. For absolute mode, the notation is @#address where address is the word that follows the instruction.

9.1.5 Autodecrement Mode

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); the register is then 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).

9.1.6 Autodecrement Deferred Mode

In autodecrement deferred mode addressing, 2 is subtracted from the contents of register n; 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).

9.1.7 Index Mode

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. 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.

9.1.8 Index Deferred Mode

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 are the address of the operand:

```
        OA = (Rn + index)
        operand = (OA)
```

If Rn denotes PC, the updated contents of the PC are used. 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.

```
  5   3 2   0
 +-----+-----+
 |mode | reg |
 +-----+-----+
```

a. Address Mode Operand Specifier

```
  2   0
 +-----+
 | reg |
 +-----+
```

b. Register Operand Specifier

```
  7              0
 +---------------+
 |    displ.bb   |
 +---------------+
```

c. Eight-Bit Displacement Branch Destination Specifier

```
  5         0
 +-----------+
 | displ.b6  |
 +-----------+
```

d. Six-Bit Displacement Branch Destination Specifier

```
  4       0
 +---------+
 |  mask   |
 +---------+
```

e. Five-Bit Literal Specifier

Figure 9-1 Compatibility-Mode Operand Specifiers

9.2 THE STACK

General register R6 is used as the stack pointer by certain instructions, as in the PDP-11 system. It is not, however, used by the hardware for any exceptions or interrupts. There is also no stack overflow protection in compatibility mode.

9.3 PROCESSOR STATUS WORD

PDP-11 compatibility mode uses a subset of the full PDP-11 processor status word. Only bits <4:0> are used; bits <15:5> are zero. When an RTI or RTT instruction is executed, bits <15:5> in the saved PSW on the stack are ignored. Compatibility-mode PSW bits <4:0> have the same meaning as do VAX PSL bits <4:0>. They are, respectively, PSL<T,N,Z,V,C>. See Chapter 1 for a description of the PSL.

9.4 INSTRUCTIONS

Table 9-1 lists the instructions provided in compatibility mode.

Table 9-2 lists the trap instructions that cause the processor to fault to native mode, where either the complete trap may be serviced or where the instruction may be simulated.

The instructions listed in Table 9-3 and all other opcodes not listed in Tables 9-1 or 9-2 are considered reserved instructions in compatibility mode. These instructions fault to native mode.

Note that no floating-point instructions are included in compatibility mode.

Figure 9-2 shows seven compatibility-mode instruction formats.

Table 9-1: Compatibility-Mode Instructions

Opcode (Octal)	Mnemonic
000002	RTI
000006	RTT
0001DD	JMP
00020R	RTS
000240--000277	Condition codes
0003DD	SWAB
000400--003777	Branches
100000--103777	Branches
004RDD	JSR
.050DD	CLR(B)
.051DD	COM(B)
.052DD	INC(B)
.053DD	DEC(B)
.054DD	NEG(B)
.055DD	ADC(B)
.056DD	SBC(B)
.057SS	TST(B)
.060DD	ROR(B)
.061DD	ROL(B)
.062DD	ASR(B)
.063DD	ASL(B)
0065SS	MFPI*
0066DD	MTPI*
1065SS	MFPD*
1066DD	MTPD*
0067DD	SXT
070RSS	MUL
071RSS	DIV
072RSS	ASH
073RSS	ASHC
074RDD	XOR
077RNN	SOB
.1SSDD	MOV(B)
.2SSSS	CMP(B)
.3SSSS	BIT(B)
.4SSDD	BIC(B)
.5SSDD	BIS(B)
06SSDD	ADD
16SSDD	SUB

Key: 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 9-2: Compatibility-Mode Trap Instructions

Opcode (Octal)	Mnemonic
000003	BPT
000004	IOT
104000--104377	EMT
104400--104777	TRAP

Table 9-3: Compatibility-Mode Reserved Instructions

Opcode (Octal)	Mnemonic
000000	HALT
000001	WAIT
000005	RESET
000007	MFPT
00023N	SPL
0064NN	MARK
0070DD	CSM
07500R	FADD--FIS
07501R	FSUB--FIS
07502R	FMUL--FIS
07503R	FDIV--FIS
076XXX	Extended Instructions
1064SS	MTPS
1067DD	MFPS
17XXXX	FP11 Floating Point

Key: R - Register specifier
SS - Source operand specifier
DD - Destination operand specifier

```
 15      12 11          6 5          0
+----------+------------+-----------+
| opcode   |   src.rx   |  dst.wx   |
+----------+------------+-----------+
```

a. Double Operand Format with Two Address Mode Specifiers

Figure 9-2 Compatibility-Mode Instruction Formats

```
 15          9 8   6 5         0
+-------------+-----+-----------+
|    opcode   | reg |  src.rw   |
+-------------+-----+-----------+
```

b. Double Operand Format with Register and Address Mode Specifiers

```
 15          9 8   6 5         0
+-------------+-----+-----------+
|    opcode   | reg | displ.b6  |
+-------------+-----+-----------+
```

c. Loop Format with Register and 6-Bit Branch Displacement Specifiers

```
 15            8 7             0
+---------------+---------------+
|     opcode    |    displ.bb   |
+---------------+---------------+
```

d. Branch Format 8-Bit Branch Displacement Specifier

```
 15                6 5         0
+-------------------+-----------+
|       opcode      |  dst.wx   |
+-------------------+-----------+
```

e. Single Operand Format with Address Mode Specifier

```
 15                      3 2   0
+-------------------------+-----+
|         opcode          | reg |
+-------------------------+-----+
```

f. Single Operand Format with Register Specifier

```
 15                            0
+-------------------------------+
|           opcode              |
+-------------------------------+
```

g. Zero Operand Format

Figure 9-2 Compatibility-Mode Instruction Formats (continued)

9.4.1 Single Operand Instructions

The following single operand instructions are described in this section. The instructions are grouped according to type: arithmetic, logical, shifts, multiprecision, and rotates.

Arithmetic:

```
     CLR(B) dst.wx

     DEC(B) dst.mx

     INC(B) dst.mx

     NEG(B) dst.mx

     TST(B) src.rx
```

Logical:

```
     COM(B) dst.mx
```

Shifts:

```
     ASR(B) dst.mx

     ASL(B) dst.mx
```

Multiprecision:

```
     ADC(B) dst.mx

     SBC(B) dst.mx

     SXT dst.ww
```

Rotates:

```
     ROL(B) dst.mx

     ROR(B) dst.mx

     SWAB dst.mw
```

CLR Clear

Format:

```
        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. The instruction is single operand format with address mode specifier. See Figure 9-2e.

DEC Decrement

Format:

```
        opcode dst.mx
```

Operation:

```
        dst <- dst - 1;
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

0053	DEC	Decrement Word
1053	DECB	Decrement Byte

Description:

One is subtracted from the destination operand, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier. See Figure 9-2e.

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:

```
        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. The instruction is single operand format with address mode specifier. See Figure 9-2e.

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:

```
        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):

0054	NEG	Negate Word
1054	NEGB	Negate Byte

Description:

The destination operand is negated (two's complement), and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier. See Figure 9-2e.

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:

```
opcode src.rx
```

Operation:

```
src - 0;
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

0057	TST	Test Word
1057	TSTB	Test Byte

Description:

The condition codes are affected according to the value of the source operand. The instruction is single operand format with address mode specifier. See Figure 9-2e.

COM Complement

Format:

```
        opcode dst.mx
```

Operation:

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

Condition Codes:

```
        N <- dst LSS 0;
        Z <- dst EQL 0;
        V <- 0;
        C <- 1;
```

Exceptions:

none

Opcodes (octal):

0051	COM	Complement Word
1051	COMB	Complement Byte

Description:

The destination operand is complemented (one's complement), and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier. See Figure 9-2e.

ASR Arithmetic Shift Right

Format:

```
        opcode dst.mx
```

Operation:

```
        dst <- dst shifted one place to the right;
```

Condition Codes:

```
        N <- dst LSS 0;
        Z <- dst EQL 0;
        V <- {bit shifted out} XOR {dst LSS 0};
        C <- bit shifted out;
```

Exceptions:

none

Opcodes (octal):

0062	ASR	Arithmetic Shift Right Word
1062	ASRB	Arithmetic Shift Right Byte

Description:

The destination operand is arithmetically shifted right by one bit and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier. See Figure 9-2e.

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:

```
        opcode dst.mx
```

Operation:

```
        dst <- dst shifted one place to the left;
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

0063	ASL	Arithmetic Shift Left Word
1063	ASLB	Arithmetic Shift Left Byte

Description:

The destination operand is arithmetically shifted left by one bit, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier. See Figure 9-2e.

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:

```
        opcode dst.mx
```

Operation:

```
        dst <- dst + C;
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

0055	ADC	Add Carry to Word
1055	ADCB	Add Carry to Byte

Description:

The contents of the condition code C-bit are added to the destination operand, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier. See Figure 9-2e.

Note:

Integer overflow occurs if the most positive integer is incremented. On overflow, the result is the most negative integer.

SBC Subtract Carry

Format:

```
        opcode dst.mx
```

Operation:

```
        dst <- dst - C;
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

0056	SBC	Subtract Carry from Word
1056	SBCB	Subtract Carry from Byte

Description:

The contents of the condition code C-bit are subtracted from the destination operand, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier. See Figure 9-2e.

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:

```
opcode dst.ww
```

Operation:

```
if N EQL 1 then dst <- -1 else dst <- 0;
```

Condition Codes:

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

Exceptions:

none

Opcode (octal):

0067 SXT Sign Extend

Description:

If the condition code N-bit is set, then the destination operand is replaced by -1; otherwise, the destination operand is cleared. The instruction is single operand format with address mode specifier. See Figure 9-2e.

Note:

If the PC is used as the destination operand, the results and the next instruction executed are UNPREDICTABLE.

ROL Rotate Left

Format:

```
        opcode dst.mx
```

Operation:

```
        dst'C <- dst'C rotated left;
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

0061	ROL	Rotate Left Word
1061	ROLB	Rotate Left Byte

Description:

The condition code C-bit and the destination operand are rotated left by one bit position; that is, the C-bit gets the most significant bit of the destination operand, and the destination is replaced by the destination shifted left by one bit with the initial C-bit filling the least significant bit. The instruction is single operand format with address mode specifier. See Figure 9-2e.

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 because of the rotate.

ROR Rotate Right

Format:

```
        opcode dst.mx
```

Operation:

```
        dst'C <- dst'C rotated right;
```

Condition Codes:

```
        N <- dst LSS 0;
        Z <- dst EQL 0;
        V <- {C bit changed due to rotate};
        C <- {bit rotated out of dst};
```

Exceptions:

none

Opcodes (octal):

0060	ROR	Rotate Right Word
1060	RORB	Rotate Right Byte

Description:

The condition code C-bit and the destination operand are rotated right by one bit position; that is, the C-bit gets the least significant bit of the destination operand, and the destination is replaced by the destination shifted right by one bit with the initial C-bit filling the most significant bit. The instruction is single operand format with address mode specifier. See Figure 9-2e.

Note:

The rotate instructions operate on the destination operand and the condition code C-bit taken as a circular datum.

SWAB   Swap Bytes

Format:

```
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

Opcode (octal):

```
0003    SWAB    Swap Bytes
```

Description:

The high and low bytes of the destination word operand are swapped. The instruction is single operand format with address mode specifier. See Figure 9-2e.

Note:

If the PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.

9.4.2 Double Operand Instructions

The following PDP-11 compatibility-mode double-operand instructions are described in this section. The instructions are grouped according to type: arithmetic and logical, and shift.

Arithmetic and Logical:

```
MOV(B) src.rx, dst.mx

ADD src.rw, dst.mw

SUB src.rw, dst.mw

CMP(B) src1.rx, src2.rx

MUL reg, src.rw

DIV reg, src.rw

XOR reg, dst.mw

BIS(B) src.rx, dst.mx

BIC(B) src.rx, dst.mx

BIT(B) src1.rx, src2.rx
```

Shift:

```
ASH reg, src.rw

ASHC reg, src.rw
```

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:

```
        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 (octal):

01	MOV	Move Word
11	MOVB	Move Byte

Description:

The destination operand is replaced by the source operand. The instruction is double operand format with two address mode specifiers. See Figure 9-2a.

Note:

The low byte is sign-extended on a MOVB to a register; that is, bits <15:8> of the destination register are replaced by bit <7> of the source operand.

ADD Add

Format:

```
        opcode src.rw, dst.mw
```

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcode (octal):

06	ADD	Add Word

Description:

The source operand is added to the destination operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers. See Figure 9-2a.

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:

```
opcode src.rw, dst.mw
```

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcode (octal):

16	SUB	Subtract Word

Description:

The source operand is subtracted from the destination operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers. See Figure 9-2a.

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:

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

Operation:

```
        tmp <- src1 - src2;
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

02	CMP	Compare Word
12	CMPB	Compare Byte

Description:

The source 1 operand is compared with the source 2 operand. The only action is to set the condition codes. The instruction is double operand format with two address mode specifiers. See Figure 9-2a.

Note:

Integer overflow occurs if the operands are of different sign and the result of the subtraction (src1 - src2) has the same sign as the source 2 operand.

MUL Multiply

Format:

```
opcode reg, src.rw
```

Operation:

```
tmp<31:0> <- Rn * src;
Rn <- tmp<31:16>;
R[n OR 1] <- tmp<15:0>;
```

Condition Codes:

```
N <- tmp LSS 0;
Z <- tmp EQL 0;
V <- 0;
C <- {result cannot be represented in 16 bits};
```

Exceptions:

none

Opcode (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. The instruction is double operand format with register and address mode specifiers. See Figure 9-2b.

Note:

1. The C-bit is set if the result of the multiplication cannot be represented in 16 bits; that is, if 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 16 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:

```
        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

Opcode (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:

```
        opcode reg, dst.mw
```

Operation:

```
        dst <- Rn XOR dst;
```

Condition Codes:

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

Exceptions:

none

Opcode (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. The instruction is double operand format with register and address mode specifiers. See Figure 9-2b.

BIS Bit Set

Format:

```
        opcode src.rx, dst.mx
```

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

05	BIS	Bit Set Word
15	BISB	Bit Set Byte

Description:

The source operand is ORed with the destination operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers. See Figure 9-2a.

BIC Bit Clear

Format:

```
opcode src.rx, dst.mx
```

Operation:

```
dst <- dst AND {NOT src};
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

04	BIC	Bit Clear Word
14	BICB	Bit Clear Byte

Description:

The destination operand is ANDed with the one's complement of the source operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers. See Figure 9-2a.

BIT Bit Test

Format:

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

Operation:

```
        tmp <- src1 AND src2;
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

03	BIT	Bit Test Word
13	BITB	Bit Test Byte

Description:

The source 1 operand is ANDed with the source 2 operand. The only action is to set the condition codes. The instruction is double operand format with two address mode specifiers. See Figure 9-2a.

ASH Arithmetic Shift

Format:

```
opcode reg, src.rw
```

Operation:

```
Rn <- Rn shifted src<5:0> bits;
```

Condition Codes:

```
N <- Rn LSS 0;
Z <- Rn 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

Opcode (octal):

072	ASH	Arithmetic Shift

Description:

The specified register is arithmetically shifted by the number of bits specified by the count operand (bits <5:0> 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. The instruction is double operand format with register and address mode specifiers. See Figure 9-2b.

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:

```
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

Opcode (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); 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. The instruction is double operand format with register and address mode specifiers. See Figure 9-2b.

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.

9.4.3 Branch Instructions

The following PDP-11 compatibility-mode branch instructions are described in this section.

```
BCC displ.bb

BCS displ.bb

BEQ displ.bb

BGE displ.bb

BGT displ.bb

BHI displ.bb

BHIS displ.bb

BLE displ.bb

BLO displ.bb

BLOS displ.bb

BLT displ.bb

BMI displ.bb

BNE displ.bb

BPL displ.bb

BR displ.bb

BVC displ.bb

BVS displ.bb

SOB reg, displ.b6
```

BR Branch

Format:

```
        opcode displ.bb
```

Operation:

```
        PC <- PC + SEXT(2*displ);
```

Condition Codes:

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

Exceptions:

none

Opcode (octal):

0004	BR	Branch

Description:

Twice the sign-extended displacement is added to the PC, and the PC is replaced by the result. The instruction is branch format with 8-bit displacement. See Figure 9-2d.

B   Branch on (condition)

Format:

```
opcode displ.bb
```

Operation:

```
if condition then PC <- PC + SEXT(2*displ);
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):		Condition	
0014	BEQ	Z EQL 1	Branch on Equal
0010	BNE	Z EQL 0	Branch Not Equal
1004	BMI	N EQL 1	Branch on Minus
1000	BPL	N EQL 0	Branch on Plus
1034	BCS, BLO	C EQL 1	Branch on Carry Set, Branch on Lower
1030	BCC, BHIS	C EQL 0	Branch on Carry Clear, Branch on Higher or Same
1024	BVS	V EQL 1	Branch on Overflow Set
1020	BVC	V EQL 0	Branch on Overflow Clear
0024	BLT	{N XOR V} EQL 1	Branch on Less Than
0020	BGE	{N XOR V} EQL 0	Branch on Greater Than or Equal
0034	BLE	{Z OR {N XOR V}} EQL 1	Branch on Less Than or Equal
0030	BGT	{Z OR {N XOR V}} EQL 0	Branch on Greater Than
1010	BHI	{C OR Z} EQL 0	Branch on Higher
1014	BLOS	{C OR Z} EQL 1	Branch on Lower or Same

Description:

The condition codes are tested and, if the condition indicated by the instruction is met, twice the sign-extended displacement is added to the PC; the PC is replaced by the result. These instructions are branch format with 8-bit displacement. See Figure 9-2d.

SOB Subtract One and Branch

Format:

```
opcode reg, displ.b6
```

Operation:

```
Rn <- Rn - 1;
if Rn NEQ 0 then PC <- PC - ZEXT(2*displ);
```

Condition Codes:

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

Exceptions:

none

Opcode (octal):

077	SOB	Subtract One and Branch

Description:

One is subtracted from the specified register, and the register is replaced by the result. If the register is not equal to zero, twice the zero-extended displacement is subtracted from the PC; the PC is replaced by the result. The instruction is loop format. See Figure 9-2c.

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.

9.4.4 Jump and Subroutine Instructions

The following PDP-11 compatibility-mode jump and subroutine instructions are described in this section.

```
JMP dst.aw

JSR reg, dst.aw

RTS reg
```

JMP Jump

Format:

```
        opcode dst.aw
```

Operation:

```
        PC <- dst;
```

Condition Codes:

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

Exceptions:

compatibility-mode illegal instruction

Opcode (octal):

0001	JMP	Jump

Description:

The PC is replaced by the destination operand. The instruction is single operand format with address mode specifier. See Figure 9-2e.

Note:

A compatibility-mode illegal instruction fault occurs if destination mode 0 is used.

JSR Jump to Subroutine

Format:

```
opcode reg, dst.aw
```

Operation:

```
tmp <- dst;         ! Value of Rn is affected by
-(SP) <- Rn;        ! dst specifier evaluation.
Rn <- PC;
PC <- tmp;
```

Condition Codes:

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

Exceptions:

compatibility-mode illegal instruction

Opcode (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. The instruction is double operand format with register and address mode specifier. See Figure 9-2b.

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:

opcode reg

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcode (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. The instruction is single operand format with register specifier. See Figure 9-2f.

9.4.5 Return from Interrupts and Traps

The following compatibility-mode return-from-interrupts and return-from-trap instructions are described in this section.

RTI

RTT

RTI	Return from Interrupt
RTT	Return from Trap

Format:

```
        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 five bits of the PSW are replaced by the corresponding bits of the second word popped from the stack. The instruction is zero operand format. See Figure 9-2g.

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.

9.4.6 Miscellaneous Instructions

The following miscellaneous compatibility-mode instructions are described in this section.

MTP{I,D} dst.ww

MFP{I,D} src.rw

NOP

CLC

CLV

CLZ

CLN

CCC

SEC

SEV

SEZ

SEN

SCC

MTP Move To Previous Space

Format:

```
        opcode dst.ww
```

Operation:

```
        tmp <- (SP)+;           !Pop source from stack (updating SP)
        dst <- tmp;             !Write source to destination
```

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

0066	MTPI	Move To Previous Instruction Space
1066	MTPD	Move To Previous Data Space

Description:

In compatibility mode, this PDP-11 instruction works like a POP instruction. The destination operand is replaced by a word popped from the stack. The instruction is single operand format with address mode specifier. See Figure 9-2e.

Note:

The implied source operand specifier is evaluated before the destination specifier.

MFP   Move From Previous Space

Format:

```
opcode src.rw
```

Operation:

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

Condition Codes:

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

Exceptions:

none

Opcodes (octal):

0065	MFPI	Move From Previous Instruction Space
1065	MFPD	Move From Previous Data Space

Description:

In compatibility mode, this PDP-11 instruction works like a PUSH instruction. The source operand is pushed onto the stack. The instruction is single operand format with address mode specifier. See Figure 9-2e.

CC Condition Code Operators

Format:

```
        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	NOP	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 one's complement of mask<3:0> and the condition codes are replaced by the result. The instruction is zero operand format. See Figure 9-2g. Bits <4:0> of the opcode are used as the mask operand.

9.5 ENTERING AND LEAVING COMPATIBILITY MODE

Compatibility mode is entered by executing an REI instruction with the compatibility-mode bit set in the PSL on the stack. Other bits in the PSL either have the effects they have in native mode or are required to have specific values in compatibility mode. PSL<TP>, <T>, <N>, <Z>, <V>, and <C> have the same effects and meanings as they have in native mode. PSL<FPD>, <IS>, <IPL>, <IV>, <FU>, <DV> must be 0, and <CUR_MOD> and <PRV_MOD> must be 3.

VAX native mode is returned to from compatibility mode by the compatibility-mode program causing an exception, or by an interrupt.

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 0, or a reserved operand fault occurs.

9.6 NATIVE-MODE AND COMPATIBILITY-MODE REGISTERS

Compatibility-mode registers R0 through R6 are bits <15:0> of VAX general registers R0 through R6, respectively. Compatibility-mode register R7 (PC) is bits <15:0> 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; 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.

9.7 COMPATIBILITY-MODE MEMORY MANAGEMENT

PDP-11 addresses are 16-bit byte addresses. Hence, compatibility-mode programs are confined to execute in the first 64K bytes of the per-process part of the virtual address space. A one-to-one correspondence exists between a compatibility-mode virtual address and its VAX counterpart. (Virtual address 0, for example, references the same location in both modes.) A compatibility mode address is interpreted in the following paragraphs as a native mode address by appending zero in bits <31:16> to the compatibility-mode address in bits <15:0>.

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 native 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 native mode can also occur when relocating a compatibility-mode address. See Chapter 4, Memory Management.

9.8 COMPATIBILITY-MODE EXCEPTIONS AND INTERRUPTS

All interrupts and exception conditions that occur while the processor is in compatibility mode cause the processor to enter native mode. These conditions are serviced as indicated in Chapter 5 (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 three-longword frame on the kernel stack containing PSL and PC, and one longword of exception-dependent 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 5, Interrupts and Exceptions, for definitions of trap, fault, and abort.)

9.8.1 Odd Address Error Abort

An odd address error abort is caused in compatibility mode whenever a word reference is attempted on a byte boundary. The code for odd address errors is 6.

9.8.2 Faults

The following paragraphs give the compatibility-mode instruction faults and their corresponding code numbers.

Reserved Instruction Fault --- A reserved instruction fault occurs for opcodes that are defined as reserved in compatibility mode (see the section "Instructions" earlier in this chapter). The code for the reserved instruction fault is 0.

BPT Instruction Fault --- The code for the BPT instruction fault is 1.

IOT Instruction Fault --- The code for the IOT instruction fault is 2.

EMT Instruction Fault --- The fault code for the group of EMT instructions is 3.

TRAP Instruction Fault --- The fault code for the group of TRAP instructions is 4.

Illegal Instruction Fault --- In compatibility mode, JMP and JSR instructions with a register destination are illegal. The fault code for illegal instructions is 5.

9.9 TRACING IN COMPATIBILITY MODE

In compatibility mode, a trace fault occurs at the beginning of an instruction when PSW<T> is set at the beginning of the prior instruction. This effect is achieved by using PSL<TP>. (See section 5.4.5 for a description of tracing.) On trace faults, a two-longword kernel stack frame is created, containing PSL and PC. IPL and IS are 0 and CM is 1 in the stacked PSL. Compatibility-mode trace fault uses the same vector as native-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 have been problems with the operation of PSW<T> on PDP-11s. PSL<TP> solves those problems. For that reason, the operation of the PSW<T> in compatibility mode is not identical to that of PDP-11s.\

There are two ways to get PSW<T> set at the beginning of a compatibility-mode instruction:

- o An RTT or RTI instruction is executed in compatibility mode with T set in the PSW image on the stack. In this case, the next instruction is executed (the 1 pointed to by the PC on the stack), and a trace fault is taken before the following instruction.

- o An REI instruction is executed in native mode which has both T 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. (See Chapter 5 for a complete description of the interaction of REI, PSL<T>, and PSL<TP>. The operations that occur as a function of these conditions are the same whether or not compatibility mode is being entered from the REI.)

PSW<T> interacts with other compatibility-mode operations as follows. For interaction with other than compatibility mode, see Chapter 5.

1. PSW<T> is set (but PSL<TP> is clear) at the beginning of any compatibility-mode instruction that does not cause a compatibility-mode fault.

   In this case, the instruction sets PSL<TP> and executes. A trace fault is taken before the next instruction. The saved PSL has T set and TP clear. The compatibility-mode executive can take one of the following courses of action:

   a. If it services the exception directly, it can clear T 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.

b. If it returns the trace exception to compatibility mode, it pushes a (16-bit) PC and (16-bit) PSW with T set on the compatibility-mode user stack to simulate the effect of the PDP-11 trace trap. It then clears T 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 T 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. PSW<T> is set (but PSL<TP> is clear) at the beginning of an RTI or RTT.

   The RTT or RTI instruction executes, and PSL<TP> is set. A trace fault occurs before the next instruction is executed. Two different cases exist depending on whether or not T was set in the image of the PSW which was popped from the stack by the RTT or RTI instruction:

   a. PSW<T> is not set. Neither TP nor T will be set in the saved PSL on the kernel stack.

   b. PSW<T> is set. PSL<TP> will not be set, and PSW<T> will be set, as is the case for other compatibility-mode instructions.

3. PSW<T> is set (but PSL<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.

9.10 UNIMPLEMENTED PDP-11 TRAPS

Several traps that occur in PDP-11 systems are not implemented in compatibility mode:

- There is no stack overflow trap. This is equivalent to the user mode of the KT11 where there is also no overflow protection. Stack overflow can be provided by the compatibility-mode executive using the memory management mechanisms.
- There is no concept of a double error trap in compatibility mode, since the first error always puts the processor in native mode.

- o All other exception conditions such as power failure, memory parity, and memory management exceptions cause the processor to enter native mode.

9.11 COMPATIBILITY-MODE I/O REFERENCES

Neither instruction stream references nor data reads or writes can be to I/O space. The results are UNPREDICTABLE if I/O space is referenced from compatibility mode.

9.12 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 native mode.

9.13 PROGRAM SYNCHRONIZATION

All PDP-11 systems 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-11 systems. In compatibility mode, instructions that have modify destinations will perform this synchronization for UNIBUS I/O device registers, if the system has a UNIBUS interconnect, and never for memory.

Compatibility-mode procedures can write data that is to be subsequently executed as an instruction without requiring any additional synchronization.

Change History:

Revision J. Rich Brunner, December 1989.

Revision H. Tim Leonard, May 1987.

Revision E. Al Thomas, September 1986.
- o Change the format of the chapter to correspond to the rest of the manual.

Revision D. Tim Leonard, March 1985.
- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Compatibility mode is optional. Operating system may emulate it.

Revision 6. Tom Eggers, 26 July 1982.
- o Define order of specifier evaluation for MTPD and MTPI.
- o Odd address abort may precede trace faults.
- o References to I/O space are UNPREDICTABLE.

Revision 5, add instruction descriptions. Dileep Bhandarkar, 20 February 1980.
- o Add instruction descriptions.
- o Add addressing mode descriptions.
- o Clarify trace fault operation.

Revision 4. Dave Cutler, 28 February 1977.
Revision 3, results of April Task Force review. Dave Cutler, 3 June 1976.

The chapter has been extensively modified due to the restartability of instructions and the new memory management architecture. Virtually all sections were affected. The following are major revisions:
- o The PDP-11 environment exclusion list was expanded to include more features that are not supported by compatibility mode hardware.
- o MFPI, MFPD, MTPI, MTPD were moved to the list of supported instructions. They execute exactly like push and pop instructions and ignore previous mode.
- o Compatibility mode can now be entered only with an REI instruction since the process structure has been deleted from the architecture. The number of privileged bits in PSL has been reduced due to the restartability of instructions.
- o Compatibility mode programs directly map into the first 64k bytes of the per process part of the address space without any special address manipulations.
- o Exceptions now push information on the kernel stack and ISL is no longer present in the architecture.
- o The new memory management architecture obviated the need for the MCMA instruction.

Revision 2. Steve Rothman, 1 January 1976.
- o The chapter has been extensively rewritten. Included now are complete definitions of the interfaces to VAX native mode,

including memory management and exceptions. A set of notes has also temporarily been added at the end of the chapter that answers some common questions, describes open issues, and explains rejected alternatives to some of the compatibility mode specifications. The other items in this list are only the changes to revision 1 of this chapter, not the additions.

- RTI and RTT no longer perform the same operations in compatibility mode. They are now exactly equivalent to their PDP-11 counterparts.
- The comment about floating-point simulation has been changed since the native mode floating-point instructions have changed.
- The Move Compatibility Mode Address instruction has been changed to make it completely general with respect to restriction level.

Revision 1. Steve Rothman, 25 September 1975.

CHAPTER 10

SYSTEM BOOTSTRAPPING AND CONSOLE

A VAX processor can be in one of five major states: attempting to load and start (bootstrap) the operating system, attempting to restart the operating system, powered off, halted, or running. This chapter describes the processor when it is not running and describes the transitions between major states.

The four major states described in this chapter are differentiated from the running state. When the processor is running, it interprets instructions, services interrupts and exceptions, and initiates I/O operations. The console acts like a normal operating system terminal (the console is in program I/O mode).

When the processor is halted, it does not interpret instructions, service interrupts or exceptions, or initiate I/O operations. The console interprets a command language that provides control over the system (the console is in console I/O mode).

When system power supplies are unable to provide power to the processor, the processor halts, and is powered off.

The console can restart a halted operating system and can also load and start (bootstrap) an operating system. How the console handles these states is described in the following sections.

10.1 SYSTEM BOOTSTRAPPING

System bootstrap can occur as the result of a powerfail recovery, a processor halt, or the operator entering a BOOT command at the console. Also, a bootstrap can occur if the operating system moves a value of 0F02 into the TXDB register. See the section on Console Terminal Registers for more information about TXDB. See the section on Major System State Transitions for a complete description of these state transitions.

To prevent repeated attempts and failures to bootstrap or restart the operating system, the console maintains two flags called the bootstrap-in-progress flag and the restart-in-progress flag. \These flags have been called the "cold start" and "warm start" flags within

DEC.\ If a system bootstrap or restart would occur automatically but the corresponding flag is already set, the console assumes that an attempt has already been made and has failed. The console therefore does not try again.

To load and start (bootstrap) the operating system, the console searches for a section of correctly functioning system memory large enough to hold a primary bootstrap program (called VMB). If a section of memory is found, the console loads and starts VMB. VMB loads and starts the operating system. \For more information about VMB, ask the VMS development group.\

The console uses this algorithm to bootstrap the operating system:

1. If this bootstrap is the result of a console BOOT command, skip to step 4.

2. Print the message "Attempting system bootstrap" on the console terminal.

3. Check to see if the bootstrap-in-progress flag is set. If so, boot fails.

4. Set the bootstrap-in-progress flag.

5. Locate a page-aligned block of good memory. VAX Processors announced after 1987 must locate a block 256-kilobytes in length; earlier VAX processors locate a block 64-kilobytes in length. \Although a 256-kilobyte block for VMB is the new requirement, VMS will continue to support a 64-kilobyte block for VMB on existing VAX processors that were announced before 1988.\

   Recall that when memory management is disabled, virtual addresses are translated by the processor to a particular range of addresses in the physical address space through an implementation-dependent mechanism. When locating the block of good memory, the processor must ensure that the entire block resides within this range of physical addresses. For example, consider an implementation with a 34-bit physical address space which translates a 32-bit virtual address to a 34-bit physical address by truncating the two high-order bits and sign-extending the remainder. Then this implementation must locate a block within the first 512 megabytes of physical memory.

   If a block cannot be found which meets the above requirements, boot fails.

   Note that in locating the block, the processor performs a test of memory which leaves the contents of memory UNPREDICTABLE.

6. Load a bootstrap program into that good memory, starting 512 bytes from the beginning. The name of the bootstrap program

is VMB.EXE. If VMB cannot be found on the load device, or if there is an error during loading, boot fails.

7. Load the general registers:

   The contents of R0 through R4 are implementation dependent, but contain information describing:

R0	Boot device's type code.
R1	Boot device's bus address.
R2	Boot device's controller information.
R3	Boot device's unit number.
R4	Reserved.
R5	Boot control parameter from the boot command.
R6	Reserved.
R7	Reserved.
R8	Reserved.
R9	Reserved.
R10	The halt PC.
R11	The halt PSL.
AP	The halt code.
FP	Reserved.
SP	Address of 512 bytes past the start of good memory.

   \The specification for registers R0 through R4 is owned and maintained by VMS development. For a copy of the latest specification, contact the VMS development manager.\

8. Start VMB at the address in SP. VMB loads and starts the operating system.

If bootstrap fails, the console prints a message reporting the failure. The message may explain the cause of the failure, or it may just report "System bootstrap failed."

If the bootstrap is successful, the operating system sends a message to the console, causing the console to clear the bootstrap-in-progress flag. See the section System Running for a description of the messages the operating system can pass to the console.

10.2 SYSTEM RESTART

The console can restart a halted operating system. To do so, the console searches system memory for the Restart Parameter Block (RPB), a data structure constructed for this purpose by the operating system. If a valid RPB is found, the console restarts the operating system at an address specified in the RPB.

The console keeps an internal flag called restart-in-progress, which it uses to avoid repeated attempts to restart a failing operating system. An additional restart-in-progress flag may be maintained by software in the RPB.

A system restart can occur as the result of a powerfail restart, or as the result of a processor halt. See the section Major System State Transitions for a complete description.

The console uses this algorithm to restart the operating system:

1. Print the message "Attempting system restart" on the console terminal.

2. Check to see if the internal restart-in-progress flag is set. If so, restart fails.

3. Set the internal restart-in-progress flag.

4. Check to see if memory has been preserved by battery backup. If not, restart fails.

5. Look for an RPB left in memory by the operating system. If none is found, restart fails.

6. Read the software restart-in-progress flag from bit <0> of the fourth longword of the RPB. If it is set, restart fails.

7. Load SP with the address of the RPB plus 512.

8. Load AP with the halt code. Load R10 with the halt PC. Load R11 with the halt PSL.

9. Start the processor at the restart address, which is read from the second longword in the RPB.

If restart fails, the console prints a message reporting the failure. The message may explain the cause of the failure, or it may just report "System restart failed."

If the restart is successful, the operating system sends a message to the console, causing the console to clear its internal restart-in-progress flag. See the section System Running later in this chapter for a description of the messages the operating system can pass to the console.

The RPB is a page-aligned control block created by the operating system, and is shown in Figure 10-1.

The console uses this algorithm to find an RPB:

1. Search for a page of memory that contains its address in the first longword. If none is found, the search for an RPB has failed. Memory is searched linearly from low physical addresses to high physical addresses.

2. Read the second longword in the page (the physical address of the restart routine). If it is not a valid physical address, or if it is 0, return to step 1. The check for 0 is necessary to ensure that a page of zeros does not pass the test for a valid RPB.

3. Calculate the 32-bit two's complement sum (ignoring overflows) of the first 31 longwords of the restart routine. If the sum does not match the third longword of the RPB, return to step 1.

4. A valid RPB has been found.

Recall that when memory management is disabled, virtual addresses are translated by the processor to a particular range of addresses in the physical address space through an implementation-dependent mechanism. When constructing the RPB, the operating system must ensure that both the entire RPB and the restart address specified within the RPB reside within this range of physical addresses.

```
+-----------------------------------------------------------------+
|                  physical address of the RPB                    | :RPB
+-----------------------------------------------------------------+
|             physical address of the restart routine             |
+-----------------------------------------------------------------+
|   checksum of the first 31 longwords of the restart routine     |
+-----------------------------------------------------------------+
|            software restart-in-progress flag (bit 0)            |
+-----------------------------------------------------------------+
```

Figure 10-1 Restart Parameter Block (RPB)

Table 10-1: Major System State Transitions

Initial State			Final State		
	Powered Off	Halted	Booting	Restarting	Running
Powered Off		A and power restored	B and power restored	C and power restored	
Halted	powerfail		BOOT command and unlocked		START or CONTINUE and unlocked
Booting	powerfail	boot fails, or D			boot succeeds
Restart	powerfail	D	restart fails		restart succeeds
Running	powerfail	A and processor halts, or D	B and processor halts	C and processor halts	

Key: A - Console is unlocked and the halt action switch is set to Halt.
B - Console is unlocked and the halt action switch is set to Boot.
C - Console is unlocked and the halt action switch is set to restart, or the console is locked.
D - Console is unlocked and the operator types CTRL/P and HALT.

10.3 SYSTEM POWERFAIL AND RECOVERY

The system requires power to operate. The system power supply conditions external power and transforms it for use by the processor. When external power fails, the power supply requests a powerfail interrupt of the processor. The power supply continues to provide power to the processor for at least 2 milliseconds after the interrupt is requested in order to allow the operating system to save state. When the power supply can no longer provide power to the processor, the processor is halted and powered off. Battery backup options are available on some processors to supply power after external power fails, to maintain the contents of main memory, and to keep system time with the time-of-day clock.

When power is restored, the console initializes itself, initializes the processor, and examines the front panel console-lock and autorestart switches. If the console is locked, it attempts a system restart; if that fails, it attempts a system bootstrap. If the console is not locked, its action is determined by the setting of the autorestart switch.

Note that when the processor loses power, its state is lost. For example, if a processor is halted when power fails, the action on power up is still determined by the front panel switches. So the system does not necessarily stay halted.

When power is restored, the processor initializes itself. There are three kinds of hardware initialization called processor initialization, system bus initialization, and power-up initialization. Processor initialization is the result of a console INITIALIZE and involves the initialization of registers internal to the processor and the console. System bus initialization is the result of a console UNJAM command and is implementation dependent. Power-up initialization affects the system as a whole. It is the result of the restoration of power, and includes a processor initialization.

The processor must be initialized after an error halt. If the processor starts running after an error halt, without an intervening processor initialization, the operation of the processor is UNDEFINED.

The effects of processor initialization and power-up initialization on the processor's state are summarized in Tables 10-2 and 10-3. Processor registers not listed in the tables have UNPREDICTABLE values after processor or power-up initialization.

Table 10-2: Effects of Processor Initialization

Processor State	Value After Processor Initialization
PSL	041F0000 (hex)
IPL	1F (hex)
ASTLVL	4
SISR	0
ICCS	<6> and <0> clear, the rest UNPREDICTABLE
RXCS	0
TXCS	80 (hex)
MAPEN	0
PME	0
ACCS	0 if no FPA; 8001 (hex) if FPA installed
cache, instruction buffer, or write buffer	empty or valid
console previous reference	physical address, longword size, address 0
KSP, ESP, SSP, USP, ISP	UNPREDICTABLE
P0BR, P0LR, P1BR, P1LR	UNPREDICTABLE
SBR, SLR	UNPREDICTABLE
PCBB	UNPREDICTABLE
SCBB	UNPREDICTABLE
translation buffer	UNPREDICTABLE
NICR, ICR	UNPREDICTABLE
TODR	unaffected
main memory	unaffected
registers R0 through PC	unaffected
halt code	unaffected
bootstrap-in-progress flag	unaffected
restart-in-progress flag	unaffected
vector registers V0 - V15	unaffected
VMR, VLR, VCR, VSAR	unaffected
VPSR, VAER	0

Table 10-3: Additional Effects of Power-Up Initialization

Processor State	Value After Power-Up Initialization
bootstrap-in-progress flag	cleared
internal restart-in-progress flag	cleared
halt code	03 (power-up)
general registers	UNPREDICTABLE
system memory	unaffected if preserved by battery backup; otherwise, UNPREDICTABLE
TODR	unaffected if preserved by battery backup; otherwise, 0
vector registers	UNPREDICTABLE
VMR, VLR, VCR, VSAR	UNPREDICTABLE

10.4 MAJOR SYSTEM STATE TRANSITIONS

The transitions between major system states are determined by the current state and by a number of variables and events, including:

- Whether power is available to the system
- The console front panel autorestart switch
- The console lock switch
- The bootstrap-in-progress flag
- The restart-in-progress flag
- Processor error halts
- The HALT instruction
- Console commands

Table 10-1 shows the actions that cause major system state transitions.

The processor follows these rules:

- If the console is not locked when power is restored or when the processor halts, enter the state selected by the console front panel autorestart switch.
- If the console is locked when power is restored or when the processor halts, attempt a system restart.
- When system restart fails, attempt a system bootstrap.
- When system bootstrap fails, halt.
- When system bootstrap or system restart succeeds, the processor starts running.
- When the processor is halted and the console is not locked, the console BOOT command causes a system bootstrap.
- When the processor is halted and the console is not locked, the operating system moving a value of 0F02 into the TXDB register causes a system bootstrap. (See the section on Console Terminal Registers for more information about TXDB.)
- When the processor is halted and the console is not locked, the console START and CONTINUE commands cause the processor to start running.
- If the console is not locked and is running or booting or restarting, typing CTRL/P followed by a HALT command at the

console halts the processor.

10.5 SYSTEM HALTED (CONSOLE I/O MODE)

\These next sections, describing the console command language, are not required of an implementation. They are included to describe the preferred implementations of each command and action. The subset console commands are used by software that talks to the console, notably AXE - the architectural exerciser, APT - automatic product test, and RD - remote diagnosis.\

Included in this section about the system-halted state are descriptions of the console; command syntax, keywords, language subsets; errors and error messages; and halt and halt messages.

10.5.1 Console

Traditionally, computers have had a panel of lights and switches on the front for processor diagnosis and for operation of standalone programs. On VAX, this function is provided by an ASCII console through which the operator controls the processor. The ASCII console may be envisioned as a virtual console processor attached to the main processor, to a console terminal, and to a console file-storage device. Note that the console processor need not be physically separate from the main processor. It may be implemented in main processor microcode, as in the VAX-11/750 computer system. The console processor interprets commands typed on the console terminal and controls the operation of the main processor.

Through the console terminal, an operator can boot the operating system, a field service engineer can maintain the system, and a system user can communicate with running programs. Sophisticated users may also use the console for developing software.

The processor can halt as the result of an operator command, a serious system error, a HALT instruction, or a powerfail recovery. (See the section Major System State Transitions earlier in this chapter for a complete description.) When the processor is halted, the operator controls the system through the console command language. The console is in console I/O mode. The console prompts the operator for input with the string of right angle brackets (>>>).

It may be possible for the operator to put the system in an inconsistent state through the use of the console commands. For example, it may be possible to use the console to set bits in MBZ fields or to set conflicting control bits. The operation of the processor in such a state is UNDEFINED.

10.5.2 Required Hardware

As a minimum, a VAX processor provides means for attaching a console terminal, indicators for displaying "power on" and "processor running", an external control that determines the console's actions on halt or power up (the "autorestart" switch), and an external control which "locks" the console. When the console is locked, it does not recognize CTRL/P as a special character in program I/O mode but passes it to the operating system, and it ignores all input in console I/O mode.

10.5.3 Special Characters

In console I/O mode, several characters have special meanings. Some of these characters are produced by pressing a single key, while others, like the control characters, are produced by pressing the character while simultaneously pressing the control key (CTRL/).

- o Carriage return -- Typing a carriage return ends a command line. No action is taken on a command until after it is terminated by a carriage return. A null line terminated by a carriage return is treated as a valid, null command. No action is taken, and the console again prompts for input. Carriage return is echoed as carriage return, line feed.

- o Rubout -- When the operator types rubout, the console deletes the character that the operator previously typed. The console echoes with a backslash (\), followed by the character being deleted. If the operator types additional rubouts, the additional characters deleted are echoed. When the operator types a non-rubout character, the console echoes another backslash, followed by the character typed. The result is to echo the characters deleted, surrounding them with backslashes. For example:

  The operator types: EXAMI;E<rubout><rubout>NE<CR>

  The console echoes: EXAMI;E\E;\NE<CR>

  The console sees the command line: EXAMINE<CR>

  The console does not delete characters past the beginning of a command line. If the operator types more rubouts than there are characters on the line, the extra rubouts are ignored. If a rubout is typed on a blank line, it is ignored.

- o CTRL/U -- The console echoes ^U and deletes the entire line. If CTRL/U is typed on an empty line, it is echoed; otherwise, it is ignored. The console prompts for another command.

- o CTRL/S -- Typing CTRL/S stops console transmissions to the console terminal until CTRL/Q is typed. Additional input between CTRL/S and CTRL/Q is buffered as input but not echoed until CTRL/Q is typed. CTRL/S typed again before the CTRL/Q is ignored. CTRL/S and CTRL/Q are not echoed.

- CTRL/Q -- Typing CTRL/Q resumes console transmissions stopped by CTRL/S. Additional typing of CTRL/Q is ignored. CTRL/S and CTRL/Q are not echoed.

- CTRL/O -- Typing CTRL/O causes the console to throw away transmissions to the console terminal until the next CTRL/O is entered. CTRL/O is echoed as ^O<CR> when it disables output; it is not echoed when it reenables output. Output is reenabled if the console prints an error message or if it prompts for a command from the terminal. Reading a command from a command file and displaying a REPEAT command do not reenable output. When output is reenabled for reading a command, the console prompt is displayed. Output is also enabled by entering program I/O mode by CTRL/P and by CTRL/C.

- CTRL/C -- Typing CTRL/C causes the console to echo ^C and to abort processing of a command. CTRL/C has no effect as part of a binary load data stream. CTRL/C reenables output stopped by CTRL/O. When CTRL/C is typed as part of a command line, the console deletes the line as it does with CTRL/U.

- CTRL/P -- If the console is in console I/O mode, CTRL/P is equivalent to CTRL/C and is echoed as ^P. If the console is in program I/O mode and is locked, CTRL/P is not echoed but is passed to the operating system like any other character. If the console is in program I/O mode and is not locked, CTRL/P is not echoed but causes the processor to enter console I/O mode. It is UNPREDICTABLE whether CTRL/P also causes the processor to halt. HALT must subsequently be typed to halt the processor.

  \CTRL/P should also return the console to console I/O mode from diagnostic mode, if a diagnostic mode is implemented.\

If an unrecognized control character is typed (a control character here means a character with an ASCII code less than 32 decimal), it is echoed as caret followed by the character with ASCII code 64 greater. For example, BEL (ASCII code 7) is echoed as ^G, since capital G is ASCII code 7+64=71. When a control character is deleted with rubout, it is echoed the same way. After echoing the control character, the console processes it as a normal character. Unless the control character is part of a comment, the command will be invalid and the console will respond with an error message.

The response of the console to characters with codes greater than 127 (decimal) is UNPREDICTABLE.

10.5.4 Command Syntax

The console accepts commands of lengths up to 80 characters. Longer commands are responded to with an error message.

\That is, commands of 80 characters length or less will be accepted, and commands of 81 characters length or greater will not be accepted. The count does not include rubouts, rubbed out characters, or the terminating carriage return.\

Commands may be abbreviated. Abbreviations are formed by dropping characters from the end of a keyword. All commands but SET may be unambiguously abbreviated to one character. SET cannot be abbreviated to less than two characters, since it then conflicts with START. The console verifies all characters typed in a command, even when they are not needed to uniquely identify the command.

Multiple adjacent spaces and tabs are treated as a single space by the console. Leading and trailing spaces and tabs are ignored.

Command qualifiers can appear after the command keyword, or after any symbol or number in the command.

All numbers (addresses, data, counts) are in hexadecimal. (Note, though, that symbolic register names include decimal digits.) Hex digits are 0 through 9, and A through F. The console does not distinguish between upper- and lowercase either in numbers or in commands. Both are accepted.

10.5.5 Command Keywords

Following is a list of processor control, data transfer, and console control command keywords. These commands are described in the next section of this chapter.

Processor Control Commands

INITIALIZE
START <address>
CONTINUE
HALT
BOOT <device>
NEXT <count>
MICROSTEP <count>
UNJAM

Data Transfer Commands

EXAMINE <address>
DEPOSIT <address> <data>
LOAD <file>
X <address> <count>

Console Control Commands

FIND
REPEAT <command>
SET <parameter> <value>
SHOW <parameter>
TEST
! <comment>
@ <file>

10.5.6 Commands

- BOOT -

The device specification is of the format "ddan," where "dd" is a two-letter device mnemonic, "a" is an optional alphabetic adapter identifier, and "n" is a one-digit unit number.

\If the BOOT command is implemented through the use of command files, the commands in the files must not be echoed.\

The console initializes the processor and starts VMB running. (See the section System Bootstrapping earlier in this chapter.) VMB boots the operating system from the specified device. The default device is implementation dependent.

Format:

BOOT [<qualifier list>] [<device>]

Qualifier:

/R5:<data>	After initializing the processor and before starting VMB, R5 is loaded with the specified data. This allows a command file containing a BOOT command or a console user to pass a parameter to VMB.

- CONTINUE -

The processor begins instruction execution at the address currently contained in the program counter. Processor initialization is not performed. The console enters program I/O mode.

The console may use memory bytes -8(ISP) through -1(ISP) in the process of starting the processor. The contents of these bytes are then UNPREDICTABLE.

- DEPOSIT -

The command deposits the data into the address specified. If no address space or data size qualifiers are specified, the defaults are the last address space and data size used in a DEPOSIT or EXAMINE command. After processor initialization, the default address space is physical memory, the default data size is long, and the default address is zero.

If the specified data is too large to fit in the data size to be deposited, the console ignores the command and issues an error response. If the specified data is smaller than the data size to be deposited, it is extended on the left with zeros.

Format:

```
DEPOSIT [<qualifier list>] <address> <data>
```

Qualifiers:

If a qualifier that is invalid for a specified address is used, the console ignores the command and issues an error response.

/B — The data size is byte. Should not be used when accessing vector register elements.

/W — The data size is word. Should not be used when accessing vector register elements, VCR, VLR, or VMR.

/L — The data size is longword. When used with vector register elements: only the low-order longword of the specified elements are accessed; and the qualifier has no effect on the address increment which is used by the "/N" qualifier and "+" and "-" symbolic addressing. Should not be used when accessing VCR, VLR, or VMR.

/Q (optional) — The data size is quadword. For DEPOSIT, the data is entered as a 64-bit hexadecimal number with no intervening spaces. For EXAMINE, the data is displayed as two longwords separated by a single space. When accessing general registers, both R[n] and R[n+1] are accessed. Example:

```
D/Q R0 0123456789ABCDEF      Loads R0 and R1.
```

Using this qualifier when accessing vector register elements or VMR is redundant but results in no error. This qualifier should not be used to access VCR, VLR, SP, or PC.

/FF (optional) The data is entered and interpreted as a F_floating value. When used with vector register elements: only the low-order longword of the specified elements are interpreted; and the qualifier has no effect on the address increment which is used by the "/N" qualifier and "+" and "-" symbolic addressing. Should not be used when accessing VCR, VLR, or VMR. Note that not all F_floating point numbers can be rendered exactly in decimal representation; when the value of all bits must be known or set exactly, access the data as a hexadecimal longword by using the /L qualifier. Example usage:

```
D/FF R0 3.6E-19
```

/DF (optional) The data is entered and interpreted as a D_floating value. When accessing general registers, both R[n] and R[n+1] are accessed. Should not be used when accessing VCR, VLR, VMR, SP, or PC. Note that not all D_floating point numbers can be rendered exactly in decimal representation; when the value of all bits must be known or set exactly, access the data as a hexadecimal quadword (by using the /Q qualifier, for example). Example usage:

```
D/DF R0 3.6E-19
```

/GF (optional) The data is entered and interpreted as a G_floating value. When accessing general registers, both R[n] and R[n+1] are accessed. Should not be used when accessing VCR, VLR, VMR, SP, or PC. Note that not all G_floating point numbers can be rendered exactly in decimal representation; when the value of all bits must be known or set exactly, access the data as a hexadecimal quadword (by using the /Q qualifier, for example). Example usage:

```
D/GF R0 3.6E301
```

/V The address space is virtual memory. All access and protection checking occur. If the access would not be allowed to a program running with the current PSL, the console issues an error message. This includes refusing odd address references if PSL<CM> is set. Virtual space DEPOSITs cause the PTE<M> bit to be set. If memory mapping is not enabled, virtual addresses are equal to physical addresses.

/P The address space is physical memory.

/I The address space is internal processor registers. These are the registers addressed by the MTPR and MFPR instructions.

/G The address space is the general register set, R0 through PC.

/M (optional) The address space is machine-dependent.

/C The address space is microcode memory.

/U (optional) The address space is console microprocessor memory.

/N:<count> The address is the first of a range. The console deposits to the first address, then to the specified number of succeeding addresses. The increment used to calculate the succeeding addresses is the data size in effect (either the default or the one specified by a qualifier on the command line). Even if the address is the symbolic address "-", the succeeding addresses are at larger addresses. The symbolic address specifies only the starting address, not the direction of succession. For repeated references to preceding addresses, use "REPEAT DEPOSIT - <data>".

When used to access vector registers (V0 - V15), the number of elements and vector registers accessed is determined by the granularity of the symbolic address. If the symbolic address merely indicates a vector register with no element range specified, such as "V0", then the console sets the address increment as the length of one vector register and accesses all the elements of successive vector registers. If the symbolic address indicates an element range, such as V0[1], then the console sets the address increment as one element and accesses the successive elements of the specified vector register. In both cases, the increment used is unaffected by a data size qualifier which controls the size of the element values displayed or entered, such as /L or /FF.

For example:

D/P/B/N:1FF 0 0	Clears the first 512 bytes of physical memory.
D/V/L/N:3 1234 5	Deposits "5" into four longwords in virtual memory.
D/N:8 R0 FFFFFFFF	Loads general registers R0 through R8.
D/N:200 - 0	Clears the previous address, then the next 512.
D/N:1 V0 FEDCBA9876543210	Loads all elements of vector register V0 and V1 with the same value.
D/N:1 V0[3E] FEDCBA9876543210	Loads elements 3E and 3F of vector register V0 with the same value.
D/L/N:1 V0[3E] 76543210	Loads the low-order longword of elements 3E and 3F of vector register V0 with the same value.

Addresses:

If conflicting address spaces are specified, the console ignores the command and issues an error response. The address may also be one of the following symbolic addresses:

PSL The processor status longword. No address space qualifier is legal. When PSL is examined, the address space is set to M (machine dependent).

PC Program counter (general register 15). The address space is set to /G.

SP The stack pointer (general register 14). The address space is set to /G.

Rn General register n. The register number is in decimal. The address space is set to /G. For example:

```
D R5 1234 is equivalent to D/G 5 1234

D R10 6FF00 is equivalent to D/G A 6FF00
```

VCR Vector Count Register. No address space qualifier is legal; when examined, the address space is set to /VE.

VLR Vector Length Register. No address space qualifier is legal; when examined, the address space is set to /VE.

VMR Vector Mask Register. No address space qualifier is legal; when examined, the address space is set to /VE.

Vn All elements of vector register n. The register number, n, is in decimal and is between 0 and 15. No address specifier is legal; when examined the address space is set to /VE.

Vn[m] Element m of vector register n. The register number, n, is in decimal and is between 0 and 15. The element number, m, is in hexadecimal and is between 0 and 3F. No address specifier is legal; when examined the address space is set to /VE.

+ Plus sign -- The location immediately following the last location referenced in an examine or deposit. For references to physical or virtual memory spaces, the location referenced is the last address, plus the size of the last reference (1 for byte; 2 for word; 4 for long and F_floating; 8 for quad, D_floating, and G_floating). For vector registers, the granularity of the symbolic address (either Vn or Vn[m]) determines whether the next element or the next vector register is accessed; this is unaffected by a data size qualifier such as "/L". (See discussion under "/N" qualifier for more detail.) For other address spaces, the address is the last addressed referenced, plus one.

- Hyphen -- The location immediately preceding the last location referenced in an EXAMINE or DEPOSIT. For references to physical or virtual memory spaces, the location referenced is the last address minus the size of this reference (1 for byte; 2 for word; 4 for long and F_floating; 8 for quad, D_floating, and G_floating). For vector registers, the granularity of the symbolic address (either Vn or Vn[m]) determines whether the preceding element or the preceding vector register is accessed; this is unaffected by a data size qualifier such as "/L". (See discussion under "/N" qualifier for more detail.) For other address spaces, the address is the last addressed referenced minus one.

* Asterisk -- The location last referenced in an EXAMINE or DEPOSIT.

@ At sign -- The location addressed by the last location referenced in an EXAMINE or DEPOSIT. Should not be used when the last location referenced is a vector register or vector register element.

Data Format:

If conflicting data sizes are specified, the console ignores the command and issues an error response.

When depositing data to a vector register element or the VMR, the default data size and format is a quadword (regardless of whether the "/Q" qualifier is supported) and is entered on the command line as a 64-bit hexadecimal number with no intervening spaces between longwords:

```
D V1[3F] 0123456789ABCDEF
D VMR AAAAAAAAAAAAAAAA
```

When accessing VCR or VLR, the default data size is a byte.

- EXAMINE -

This command examines the contents of the specified address. If no address is specified, the plus sign (+) is assumed. The same qualifiers may be used on EXAMINE as may be used on DEPOSIT. The address may also be one of the symbolic addr sses described under DEPOSIT.

Format:

```
EXAMINE [<qualifier list>] [<address>]
```

Response:

```
        <tab><address space identifier> <address> <data>
```

The address space identifier can be:

P Physical memory. Note that when virtual memory is examined, the address space and address in the response are the translated physical address.

G General register.

I Internal processor register.

M Machine-dependent address space. When the PSL is examined, the address space identified is machine dependent.

C Microcode memory.

U (optional) Console microprocessor memory.

VE Vector Processor data: used for vector register elements, VCR, VLR, and VMR.

Displaying Data:

Quadword data is displayed as two longwords separated by a single space:

```
           let:      R0 = 89ABCDEF  and   R1 = 01234567
           then:     E/Q R0
           returns:  G  R0  01234567 89ABCDEF
```

Multiple vector register elements are displayed in pairs as:

```
        VE  Vn[m]  <data>   Vn[m+1]  <data>
```

An entire vector register is displayed as:

```
        VE  Vn[00]  <data>   Vn[01]  <data>
                           .
                           .
                           .
        VE  Vn[3E]  <data>   Vn[3F]  <data>
```

VCR, VLR, and VMR are displayed as in this example:

```
        VE  VLR  <data>
```

- FIND -

The console searches main memory starting at address zero for a page-aligned, 64 or 256-kilobyte block of good memory, or a restart parameter block (RPB). If the block is found, its address plus 512 is left in SP. If the block is not found, an error message is issued, and the contents of SP are UNPREDICTABLE. If no qualifier is specified, /RPB is assumed.

Format:

FIND [<qualifier list>]

Qualifiers:

/MEMORY Search memory for a page-aligned block of good memory. VAX processors announced after 1987 must search for a block 256-kilobytes in length; earlier VAX processors search for a block 64-kilobytes in length. Since the search may include a read and write test of memory, the search leaves the contents of memory UNPREDICTABLE.

/RPB Search memory for a restart parameter block. See the section System Restart earlier in this chapter for the search algorithm. The search leaves the contents of memory unchanged.

- HALT -

The processor stops execution of macroinstructions after completing the current macroinstruction. Neither processor initialization nor I/O initialization occurs, so I/O operations already in progress are unaffected. If the processor is already halted, the HALT command has no affect.

Response: PC = <PC>

If the processor is already halted, the response is preceded by a halt message.

Message: Already halted

- INITIALIZE -

A processor initialization is performed. See the section System Powerfail and Recovery for initial register contents.

- LOAD -

The console loads data from the specified file into memory. If no qualifiers are specified, data is loaded into physical memory starting at address 0. If an unrecoverable device or memory error occurs during the load, the command is aborted and the console issues an error message.

Format:

LOAD [<qualifier list>] <file>

Qualifiers:

/S:<address>	The data is loaded starting at the specified address.
/C	The data is to be loaded into microcode memory.
/U (optional)	The data is to be loaded into console microprocessor memory.

- MICROSTEP -

The console causes the processor to execute the specified number of microinstructions. If no count is specified, 1 is assumed. After the last microinstruction is executed, the console enters space-bar-step mode.

In space-bar-step mode, the console executes one microinstruction each time the operator presses the space bar. If the operator presses any other key, the console exits space-bar-step mode, then processes the character typed. Typing carriage return is the suggested means of exiting from space-bar-step mode.

The operator can use the NEXT command to cause the console to finish the macroinstruction executing.

Format:

MICROSTEP [<count>]

Response: uPC = <uPC>

- NEXT -

The console causes the processor to execute the specified number of macroinstructions. If no count is specified, 1 is assumed. After the last macroinstruction is executed, the console enters space-bar-step mode.

In space-bar-step mode, the console executes one macroinstruction each time the operator presses the space bar. If the operator presses any other key, the console exits space-bar-step mode, then processes the character typed. Typing carriage return is the suggested means of exiting from space-bar-step mode.

The NEXT command can be used to finish a macroinstruction partially executed by MICROSTEP. This partial execution is counted by NEXT as though it were the execution of a full instruction.

Format:

```
NEXT [<count>]

Response:       PC = <PC>
```

- REPEAT -

The console repeatedly displays and executes the specified command. The repeating is stopped when the operator types CTRL/C. Any valid console command may be specified for this command with the exceptions of the REPEAT command and the @ command. If the command is REPEAT or @, the results are UNPREDICTABLE.

Format:

```
REPEAT <command>

Response:       <dependent upon command specified>
```

- SET -

Sets the console parameter to the indicated value. The console parameters and their meanings are all implementation dependent.

Format:

```
SET <parameter> <data>
```

- SHOW -

Shows the value of the indicated console parameter. The console parameters and their meanings are all implementation dependent.

Format:

SHOW <parameter>

- START -

The console starts instruction execution at the specified address. The default address is implementation dependent. If no qualifier is present, macroinstruction execution is started. If memory mapping is enabled, macroinstructions are executed from virtual memory. The START command is equivalent to a DEPOSIT to PC followed by a CONTINUE. No INITIALIZE is performed.

The console may use memory bytes -8(ISP) through -1(ISP) in the process of starting the processor. The contents of these bytes are then UNPREDICTABLE.

Format:

START [<qualifier list>] [<address>]

Qualifiers:

/C	Microinstruction (rather than macro) execution is started.
/U (Optional)	Console microprocessor instruction execution is started.

- TEST -

The console executes a self-test. All qualifiers are optional.

Format:

TEST [<qualifier list>]

- UNJAM -

A system bus initialization is performed. The effects of a system bus initialization are implementation dependent.

- X (Binary Load and Unload Command) -

The X command is for use by automatic systems communicating with the console. It is not intended for use by operators. The console loads or unloads (that is, writes to memory or reads from memory) the specified number of data bytes, starting at the specified address. If no qualifiers specify otherwise, data is transferred to or from physical memory.

If bit <31> of the count is clear, data is to be received by the console and deposited into memory. If bit <31> of the count is set, data is to be read from memory and sent by the console. The remaining bits in the count are a positive number indicating the number of bytes to load or unload.

The console accepts the command upon receiving the carriage return. The next byte the console receives is the command checksum, which is not echoed. The command checksum is verified by adding all command characters, including the checksum (but not including the terminating carriage return or rubouts or characters deleted by rubout), into an 8-bit register initially set to zero. If no errors occur, the result is zero. If the command checksum is correct, the console responds with the input prompt and either sends data to the requester or prepares to receive data. If the command checksum is in error, the console responds with an error message. The intent is to prevent inadvertent operator entry into a mode where the console is accepting characters from the keyboard as data with no escape sequence possible.

If the command is a load (bit <31> of the count is clear), the console responds with the input prompt, then accepts the specified number of bytes of data for depositing to memory and an additional byte of received data checksum. The data is verified by adding all data characters and the checksum character into an 8-bit register initially set to zero. If the final contents of the register are non-zero, the data or checksum is in error, and the console responds with an error message.

If the command is a binary unload (bit <31> of the count is set), the console responds with the input prompt followed by the specified number of bytes of binary data. As each byte is sent, it is added to a checksum register initially set to zero. At the end of the transmission, the two's complement of the low byte of the register is sent.

If the data checksum is incorrect on a load, or if memory errors or line errors occur during the transmission of data, the entire transmission is completed and then the console issues an error message. If an error occurs during loading, the contents of the memory being loaded are UNPREDICTABLE.

If the console implements SET TERMINAL ECHO and SET TERMINAL NOECHO commands, the state of the echo flag is unaffected by the X command. Regardless of the flag, echo is suppressed while data string and checksums are being received.

It is possible to control the console through the use of the console control characters (CTRL/C, CTRL/S, CTRL/O, for example) during a binary unload. It is not possible during a binary load because all received characters are valid binary data.

Data being loaded with a binary load command must be received by the console at a rate of at least one byte per second. If the console does not receive a data byte for more than one second, the console aborts the transmission by issuing an error message and prompting for input.

The entire command, including the checksum, may be sent to the console as a single burst of characters at the console's specified character rate. To make this command useful in automated systems, the console is able to receive at least 4K bytes of data in a single X command.

NOTE TO IMPLEMENTORS

> \The command checksum is sent with the command, before waiting for a response from the console. Since it may be a control character, the console must stop processing incoming characters after receiving carriage return, until it has determined if the command is an X command. Otherwise, the checksum character can be misinterpreted by the console character input routine.\

Format:

X [<qualifier list>] <address> <count> <CR> <checksum>

Qualifiers:

/P	Data is to be read from or written to physical memory.
/C	Data is to be read from or written to microcode memory.
/U (optional)	Data is to be read from or written to console microprocessor memory.

- @ (The Indirect Command) -

The console reads and executes commands from the specified file. The commands are displayed on the console terminal as they are read. When a BOOT, START, or CONTINUE command is executed, putting the console into program I/O mode, command file processing is suspended. If a "software done" message is received by the console (see the section System Running later in this chapter) and the processor halts, command file processing is continued. If the processor halts before a "software done" message is received by the console, the remainder of the command file is ignored.

Command files can be chained by using another @ command as the last command in a file. If an @ command is encountered in the middle of a command file, the console executes it but may ignore the remainder of the original command file. It is an implementation option whether or not the console resumes execution of the original command file on completion of the secondary.

Format:

```
@ <file>
```

The comment

The comment is ignored.

Format:

```
!  <comment>
```

10.5.7 Command Language Subsets

To reduce cost, some implementations may not implement the full console command set. A subset implementation is defined.

The commands supported by a subset console are as follows:

- BOOT <device>
- CONTINUE
- DEPOSIT <address> <data>
- EXAMINE [<address>]
- INITIALIZE
- HALT
- START <address>
- TEST
- X <address> <count>
- ! <comment>

EXAMINE and DEPOSIT support the qualifiers /B, /W, /L, /P, /V, /I, /G, and the symbolic address PSL.

The control characters supported are carriage return, CTRL/P, CTRL/S, CTRL/Q, CTRL/U, and rubout.

The subset console may perform range checking on addresses and data. If it does not, it truncates values that are too large and uses the lower digits.

The subset console may accept only abbreviated commands. It may also limit the command length to less than 80 characters. It may accept only uppercase commands. Automatic systems communicating with a console must limit themselves to the commands in the subset, must abbreviate all commands, and must use only uppercase if they are to communicate with any console implementation.

10.5.8 Vector Support

Support for console access of vector registers, VCR, VLR, and VMR is required by a VAX system which implements the VAX vector processor. This support requires all vector-related features as described in section 10.5.6 with the exception of the /Q, /FF, /DF, and /GF qualifiers.

10.5.9 Options

Some features are optional, such as the diagnosis mode and the /M and /U qualifiers. The /Q, /FF, /DF, and /GF qualifiers are also optional to implement and may be implemented separately. These features may be implemented by any console, even by a subset.

10.5.10 Errors and Error Messages

The console can issue error messages in response to commands. The case (uppercase or lowercase) is implementation dependent.

The console responds to all commands within 1 second. If the processor does not respond to a console request, the console issues an error message within 1 second.

The following three messages indicate failure of the requested operation. Some implementations may abbreviate some or all of these messages to "Can't."

Can't power up
The console microprocessor cannot complete its own power-up initialization. The state of the console and that of the processor is UNDEFINED.

File not found
The file specified in a BOOT, LOAD, or @ command cannot be found.

Reference not allowed
The requested reference would violate virtual memory protection, or the address is not mapped, or the reference is invalid in the specified address space, or the value is invalid in the specified destination.

The messages below are responses to ill-formed commands. Some implementations may abbreviate some or all of these messages to "Illegal command."

Illegal command
The command string cannot be parsed.

Invalid digit
A number has an invalid digit.

Line too long
The command was too large for the console to buffer. The message is issued only after receipt of the terminating carriage return.

Illegal address
The address specified falls outside the limits of the address space.

Value too big
The value specified does not fit in the destination.

Conflicting switches
For example, two different data sizes are

specified with an EXAMINE command.

Unknown switch   The switch is unrecognized.

Unknown symbol   The symbolic address in an EXAMINE or DEPOSIT is unrecognized.

The following message is produced when a binary transfer command is improperly specified.

Incorrect checksum   The command or data checksum of an X command is incorrect. If the data checksum is incorrect, this message is issued and is not abbreviated to "Illegal command."

The following message is produced when a HALT command is given to the console and the processor is already halted.

Already halted   The operator entered a HALT command and the processor was already halted.

Some console commands may result in errors. For example, if a memory error occurs as the result of a console command, the console will respond with an error message. Such errors do not affect the halted program. Specifically, the processor stays halted, and if it is started later, no exception or interrupt occurs as the result of the console error.

10.5.11 Halts and Halt Messages

Whenever the processor halts, the console prints the response "PC = "<PC>. Except when the halt was requested by a console HALT command or by a NEXT command, the response is preceded by a halt message. For example:

```
?06     HALT executed
        PC = 800050D3
```

The number preceding the halt message is the halt code, and is passed to the operating system on a restart. Halt code 03 does not have a corresponding message. It is passed by the console during powerfail restart.

The halt messages are:

?00 CPU halted   The operator entered a HALT command while the processor was running, so the console halted the processor.

?01 Microverify complete   The console quick-verify completed successfully.

?02 CPU halted   The operator typed CTRL/P while the console was in program I/O mode. The console was not locked, and the console halted the processor.

03

Halt code 03 does not appear in a halt message but is passed by the console on powerfail restart.

?04 I-stack not valid

In attempting to push state onto the interrupt stack during an interrupt or exception, the processor discovered that the interrupt stack was mapped NO ACCESS or NOT VALID.

?05 CPU double error

The processor attempted to report a machine-check to the operating system, and a second machine-check occurred.

?06 HALT executed

The processor executed a HALT instruction in kernel mode.

?07 Invalid SCB vector

The vector had bits <1:0> set.

?08 No user WCS

An SCB vector had bits <1:0> equal to 2, and no user writable control store was installed.

?09 Error pending on halt

The processor was halted (by CTRL/P) before it could perform an error halt.

?0A CHM from I-stack

A change mode instruction was executed when PSL<IS> was set.

?0B CHM to interrupt stack

The exception vector for a change mode had bit <0> set.

?0C SCB read error

A hard memory error occurred while the processor was trying to read an exception or interrupt vector.

10.6 SYSTEM RUNNING (PROGRAM I/O MODE)

When the processor is running, the console is in program I/O mode. In this mode, all terminal interaction is handled by the operating system. The console terminal becomes like any other operating system terminal and passes through all characters (except for CTRL/P). If the console is locked, even CTRL/P is passed through. If the console is not locked, CTRL/P causes the processor to halt and the console to enter console I/O mode.

10.6.1 Console Terminal Registers

The console is accessed by the operating system through four internal processor registers. Two are associated with passing information from the console to the processor (receive registers) and two with passing information from the processor to the console (transmit registers). In each direction, there is a control and status register and a data buffer register. The registers are shown in Figure 10-2. The fields of the registers are described in Tables 10-4 through 10-7.

```
 31                                            8 7 6 5         0
+-----------------------------------------------+-+-+-----------+
|                       MBZ                     |R|I|    MBZ    |
+-----------------------------------------------+-+-+-----------+
```

a. Console Receive Control and Status Register (RXCS)

```
 31                         16 15 14  12 11    8 7             0
+-----------------------------+--+------+-------+---------------+
|               MBZ           | E| MBZ  |  ID   |     data      |
+-----------------------------+--+------+-------+---------------+
```

b. Console Receive Data Buffer Register (RXDB)

```
 31                                            8 7 6 5         0
+-----------------------------------------------+-+-+-----------+
|                       MBZ                     |R|I|    MBZ    |
+-----------------------------------------------+-+-+-----------+
```

c. Console Transmit Control and Status Register (TXCS)

```
 31                                    12 11    8 7            0
+---------------------------------------+-------+--------------+
|                  reserved             |   ID  |    data      |
+---------------------------------------+-------+--------------+
```

d. Console Transmit Data Buffer Register (TXDB)

Figure 10-2 Console Terminal Registers

Table 10-4: Fields of the RXCS Register

Name	Extent	Description
ready	<7>	Read Only. Cleared by processor initialization and by reading RXDB. When Ready is clear, RXDB is UNPREDICTABLE. When Ready is set, RXDB contains valid data to be read.
interrupt enable	<6>	Read/write. Cleared by processor initialization and by being written zero. If interrupt enable is set by software while RXDB Ready is already set, or if ready is set by the console while Interrupt enable is already set, then an interrupt is requested at IPL 14 (hex). That is, an interrupt is requested whenever the function {interrupt enable AND ready} changes from 0 to 1.

Table 10-5: Fields of the RXDB Register

Name	Extent	Description
error	<15>	An error occurred while receiving data, such as data overrun or loss of carrier. Cleared by processor initialization and by reading from RXDB.
identification	<11:8>	If zero, then data is from the console terminal. If nonzero, then the rest of the register is implementation dependent. Cleared by processor initialization and by reading from RXDB.
data	<7:0>	Data from the console terminal (if ID is zero). UNPREDICTABLE unless RXCS ready is set.

Table 10-6: Fields of the TXCS Register

Name	Extent	Description
ready	<7>	Read only. Set by processor initialization. Ready is clear when the console terminal is busy writing a character written to TXDB. Ready is set when the console terminal is ready to receive another character.
interrupt enable	<6>	Read/write. Cleared by processor initialization and by being written clear. If interrupt-enable is set when ready becomes set, or if interrupt-enable is set by software when ready is already set, an interrupt is requested at IPL 14 (hex). That is, an interrupt is requested whenever the function {interrupt enable AND ready} changes from 0 to 1.

Table 10-7: Fields of the TXDB Register

Name	Extent	Description
ID	<11:8>	If ID is written zero when TXDB is written, the data goes to the console terminal. If ID is written with 0F (hex), the data is a message to be sent to the console. If ID is neither zero nor 0F (hex), the meaning is implementation dependent.
data	<7:0>	If ID is zero, the data is a character sent to the console terminal to type. If ID is 0F (hex), the data is a message to be sent to the console, with the following meaning: 1. Software done -- A program started by a console indirect command file is signaling successful completion. When the processor halts, the console should resume processing the indirect command file. 2. Boot processor -- The console should initiate a system bootstrap. 3. Clear "restart in progress" flag -- A system restart has successfully completed. If a system restart would occur automatically, the attempt should be allowed. 4. Clear "bootstrap in progress" flag -- A system bootstrap has successfully completed. If a system bootstrap would occur automatically, the attempt should be allowed.

Change History:

Revision J. Rich Brunner, December 1989.

- Specified state of vector registers, vector control registers, and vector IPRs after processor initialization and power-up initialization.
- Add that all announced VAX processors as of 1988 must support a VMB page aligned block of 256-KB.
- Added vector support to Deposit and Examine and new optional floating point qualifiers. Restructured console command descriptions for neatness.

Revision H. Tim Leonard, May 1987.
Revision F. Al Thomas, November 1986.

- Add that the console may use memory bytes -8(ISP)..-1(ISP) in the process of starting the processor.

Revision E. Al Thomas, September 1986.

- Note in describing boot that it can be caused by moving 0F02 into TXDB.
- Clarify syntax of BOOT console command.
- Describe contents of R0 through R4 during boot.
- Change the revision number to correspond to DEC Standard 032 rev number.

Revision 3, approved. Tim Leonard, 13 September 1982.
Revision 2, restructure and add power up. Tim Leonard, 28 June 1982.
Revision 1. Tim Leonard, 14 May 1981.

CHAPTER 11

IMPLEMENTATION OPTIONS

Some parts of the VAX architecture may be included as standard features of a processor, provided as options to a processor, or omitted completely from a processor.

The implementation-option rules for the architecture reflect the need to be able to trade-off manufacturing cost, software development cost, and performance of VAX processors. The following conflicting hardware and software goals influenced the design of these rules:

- o Hardware goal -- Permit an implementor of a low-end processor to omit instructions and other features in order to reduce manufacturing cost without losing the ability to run all of the system software. The decision not to implement some options may impact the performance of certain software products.

- o Software goal -- Provide as small a number of classes of processor instruction sets as possible to reduce software development costs. In particular, a single version of each compiler or other layered software product should run on all processors in the VAX family. Also the combination of hardware and instruction emulation routines in operating systems must (as required) give the appearance of a complete architecture on all processors.

11.1 INSTRUCTION-SET OPTIONS

There are three categories for instructions that define how the instruction set may be implemented. After 1986, VAX processors must implement instructions in the base instruction group. Instructions in either an application extension group or the emulated-only instruction group may not be implemented, using the rules stated in this section.

11.1.1 Base Instruction Group

All 242 instructions in the base instruction group must be implemented in VAX processors announced after 1986. Wherever possible, VAX processors will optimize the performance of instructions in this group. VAX base system software (operating systems, runtime libraries, utilities, and code generated by DEC compilers) will target instruction use to this group. Instructions in the base group are:

Eighty-nine integer arithmetic and logical instructions: ADAWI, ADD{B,W,L}{2,3}, ADWC, ASH{L,Q}, BIC{B,W,L}{2,3}, BIS{B,W,L}{2,3}, BIT{B,W,L}, CLR{B,W,L,Q}, CMP{B,W,L}, CVTB{W,L}, CVTW{B,L}, CVTL{B,W}, DEC{B,W,L}, DIV{B,W,L}{2,3}, EDIV, EMUL, INC{B,W,L}, MCOM{B,W,L}, MNEG{B,W,L}, MOV{B,W,L,Q}, MOVZ{BW,BL,WL}, MUL{B,W,L}{2,3}, PUSHL, ROTL, SBWC, SUB{B,W,L}{2,3}, TST{B,W,L}, XOR{B,W,L}{2,3}.

Eight address instructions: MOVA{B,W,L,Q}, PUSHA{B,W,L,Q}.

Seven variable-length bit field instructions: CMPV, CMPZV, EXTV, EXTZV, FF{S,C}, INSV.

Thirty-nine branch and control instructions: ACB{B,W,L}, AOBLEQ, AOBLSS, BLSS, BLEQ, BEQL, BNEQ, BGEQ, BGTR, BLSSU, BLEQU, BGEQU, BGTRU, BVS, BVC, BB{S,C}, BB{S,C}{S,C}, BB{SS,CC}I, BLB{S,C}, BR{B,W}, BSB{B,W}, CASE{B,W,L}, JMP, JSB, RSB, SOBGEQ, SOBGTR.

Three procedure call instructions: CALLG, CALLS, RET.

Six queue instructions: INSQHI, INSQTI, INSQUE, REMQHI, REMQTI, REMQUE.

Eight character string instructions: MOVC3, MOVC5, CMPC3, CMPC5, LOCC, SKPC, SPANC, SCANC.

Twelve instructions for use by operating systems: PROBE{R,W}, CHM{K,E,S,U}, HALT, REI, LDPCTX, SVPCTX, MTPR, MFPR.

Nine miscellaneous instructions: BI{C,S}PSW, BPT, INDEX, MOVPSL, NOP, POPR, PUSHR, XFC.

Nineteen F_floating instructions: MOVF, MNEGF, CVTF{B,W,L}, CVT{B,W,L}F, CMPF, TSTF, ADDF2, ADDF3, SUBF2, SUBF3, MULF2, MULF3, DIVF2, DIVF3, CVTRFL.

Twenty-one D_floating instructions: MOVD, MNEGD, CVTD{B,W,L,F}, CVT{B,W,L,F}D, CMPD, TSTD, ADDD2, ADDD3, SUBD2, SUBD3, MULD2, MULD3, DIVD2, DIVD3, CVTRDL.

Twenty-one G_floating instructions: MOVG, MNEGG, CVTG{B,W,L,F}, CVT{B,W,L,F}G, CMPG, TSTG, ADDG2, ADDG3, SUBG2, SUBG3, MULG2, MULG3, DIVG2, DIVG3, CVTRGL.

11.1.2 Application-Extension Instruction Groups

Instructions in an application extension group may be implemented or omitted only as a group. That is, if any instruction in one of the application groups is implemented (omitted), all instructions in that application extension group are implemented (omitted). All VAX processors announced after 1986 have the option to implement zero or more of the application extension groups. Emulation software is required for instructions in unsupported application groups. VAX compilers will support features to complement the application extension groups implemented in VAX processors. The application extension groups are:

- a packed-decimal-string group with sixteen instructions: MOVP, CMPP3, CMPP4, ADDP4, ADDP6, SUBP4, SUBP6, CVTLP, CVTPL, CVTPT, CVTTP, CVTPS, CVTSP, ASHP, MULP, DIVP.

  If an instruction in this group is omitted by a processor, execution of the instruction results in an instruction-emulation exception.

- an extended-accuracy group with twenty-nine H_floating and octaword instructions: MOVH, MNEGH, CVTH{B,W,L,F,D,G}, CVT{B,W,L,F,D,G}H, CMPH, TSTH, ADDH2, ADDH3, SUBH2, SUBH3, MULH2, MULH3, DIVH2, DIVH3, CVTRHL, MOVO, CLRH (CLRO), MOVAH (MOVAO), PUSHAH (PUSHAO).

  If an instruction in this group is omitted by a processor, execution of the instruction results in a reserved-instruction fault.

11.1.3 Emulated-Only Instruction Group

No VAX processor announced after 1986 is required to implement instructions in the emulated-only instruction group. VAX processors may implement some or all of these instructions, but are not required to implement any. Software is required to support emulation of instructions in this group. VAX software-development organizations are responsible for making the business decisions around removing the use, or generation of, emulated-only instructions in their products, and take responsibility for the performance consequences of these decisions. Instructions in the emulated-only instruction group are:

- five string instructions: MATCHC, MOVTC, MOVTUC, CRC, EDITPC.

  If an instruction in this group is omitted by a processor, execution of the instruction results in an instruction-emulation exception.

- twelve floating-point instructions: ACB{F,D,G,H}, EMOD {F,D,G,H}, POLY{F,D,G,H}.

If an instruction in this group is omitted by a processor, execution of the instruction results in a reserved-instruction fault.

11.1.4 PDP-11 Compatibility Mode

No VAX processor announced after 1983 is required to implement PDP-11 compatibility mode. In a processor without compatibility mode, the execution of an REI instruction attempting to enter compatibility mode results in a reserved-operand fault.

11.2 OTHER IMPLEMENTATION OPTIONS

Certain other features of the VAX architecture are optional in VAX processor implementations.

11.2.1 Internal Processor Registers

The registers described below are optional for VAX processors announced after 1983. If any of the registers named on one of the following lines is included, all the registers on that line must be included.

- o Interval timer registers: NICR, ICR, ICCS except for <IE>. (That is, implementation of ICCS<IE> is required.)
- o Time-of-Year clock register: TODR
- o Console registers: RXCS, RXDB, TXCS, TXDB
- o Performance Monitor Enable register: PME

11.2.2 Virtual-Machine Support

This consists of support for virtual machines. If support for virtual machines is omitted, the PROBEVMR and PROBEVMW instructions generate privileged-instruction faults, the VMPSL internal processor register is omitted, and the <VM> field of the PSL register is always clear. \Parts of the specification saying things such as, "if PSL<VM> EQLU 1 then...," can clearly be omitted as well, since PSL<VM> will always be clear.\ Support for virtual machines may be included or omitted only as a whole.

11.2.3 Memory Management

The VAX memory management includes a PTE format with a 21-bit page-frame number (PFN), a hardware-maintained modify bit, and physical addresses in the SCBB, SBR, and PCBB registers. VAX processors announced after 1987 may support a PTE format with a 25-bit PFN, a software-maintained modify bit, and virtual addresses in the SCBB and PCBB registers. See Chapter 4, Memory Management, for more detail.

11.2.4 rtVAX Memory Management

The rtVAX is a variant of the VAX Architecture. An rtVAX processor implements the per-process page tables in physical memory. All other VAX processors implement the per-process page tables in virtual memory. Therefore, translation of process-space addresses is different on an rtVAX than as described in Chapter 4, Memory Management.

On an rtVAX processor, the process-space page tables are located in physical memory, and the P0BR and P1BR registers contain physical addresses.

A process-space address translation that causes a translation-buffer miss will cause one memory reference for the process PTE. On an rtVAX, this reference is to physical memory, and cannot experience further memory-mapping problems.

If part or all of either process-space page table is mapped into I/O space, into nonexistent memory, or beyond the end of the physical address space while memory mapping is enabled, the operation of the processor is UNDEFINED.

Figure 11-1 illustrates the process-space mapping registers, and Figure 11-2 illustrates virtual-to-physical address translation for process space on an rtVAX.

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 the base address of the P0PT, which in a (non-rtVAX) VAX is a virtual address in the system region, and in an rtVAX is a physical address. Figure 11-1a illustrates the P0BR. The page table entry addressed by the P0BR maps the first page of the P0 region of the virtual address space, that is, virtual byte address 0. The P0LR contains the size of the P0PT in longwords, that is, the number of page table entries. Figure 11-1b illustrates the P0LR.

The PTEs in the P0PT contain the mapping information, or point to the mapping information in the global page table if the PTE is in GPTX format. (See section 7.5.1, I/O-Device Use of PTEs.)

Writing P0LR bits <26:24> has no effect. P0LR bits <26:24> read as zero. At processor initialization time, the contents of both registers are UNPREDICTABLE.

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.

The algorithm an rtVAX uses to generate a physical address from a P0-region virtual address is as follows:

```
        PROC_PA = (P0BR+4*P0VA<29:9>)<PFN>'P0VA<8:0>     ! P0 Region
```

The P1 region of the address space is mapped by the P1 page table (P1PT). P1PT is defined by the P1 base register (P1BR) and the P1 length register (P1LR). Because P1 space grows toward 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. The P1BR contains the base address of the P1PT, which in a (non-rtVAX) VAX is a virtual address in the system region, and in an rtVAX is a physical address. Figures 11-1c and d illustrate the P1 base register and P1 length register. Note that P1LR contains the number of nonexistent PTEs. P1BR contains the address of what would be the PTE for the first page of P1, that is, virtual byte address 40000000 (hex).

The PTEs in the P1PT contain the mapping information, or point to the mapping information in the GPT if the PTE is in GPTX format. (See section 7.5.1, I/O-Device Use of PTEs, for a description of the GPTX format.)

At processor initialization time, the contents of both registers are UNPREDICTABLE. Writing P1LR<31> has no effect. The bit always reads as 0.

The algorithm an rtVAX uses to generate a physical address from a P1-region virtual address is as follows:

```
        PROC_PA = (P1BR+4*P1VA<29:9>)<PFN>'P1VA<8:0>     ! P1 Region
```

```
 3
 1                                           9 8               0
+---------------------------------------------+-----------------+
|  physical base address of the P0 page table |       MBZ       |
+---------------------------------------------+-----------------+
```

a. rtVAX P0 Base Register (P0BR)

```
 3       2 2   2 2 2 2
 1       7 6   4 3 2 1                                         0
+---------+-----+---+-------------------------------------------+
|   MBZ   | IGN |MBZ|      length of the P0PT in longwords      |
+---------+-----+---+-------------------------------------------+
```

b. P0 Length Register (P0LR)

```
 3
 1                                           9 8               0
+---------------------------------------------+-----------------+
|  physical base address of the P1 page table |       MBZ       |
+---------------------------------------------+-----------------+
```

c. rtVAX P1 Base Register (P1BR)

```
 3 3                2 2
 1 0                2 1                                        0
+-+------------------+------------------------------------------+
|I|       MBZ        | 2**21 minus length of P1PT in longwords  |
+-+------------------+------------------------------------------+
```

d. P1 Length Register (P1LR)

Figure 11-1 rtVAX Process-Space Mapping Registers

```
                       31 30 29                 9 8      0
process virtual       +-----+--------------------+--------+
address:              | 0 x | virtual page number|  byte  |
                      +-----+--------------------+--------+
                            |     extract and    |        |
                            |     check length   |        |
                  31      23|22                 2|10      |
                 +----------+--------------------+--+     |
                 |    0     | virtual page number|00|     |
                 +----------+--------------------+--+     |
                                                          |
                                  add                     |
                                                          |
                   +--------------------------------+     |
PxBR:              |  physical base address of PxPT |     |
                   +--------------------------------+     |
                                                          |
                                  yields                  |
                                                          |
                   +--------------------------------+     |
                   |    physical address of PxPTE   |     |
                   +--------------------------------+     |
                                                          |
                                  fetch                   |
                31       21 20                 0          |
               +-----------+--------------------+         |
PxPTE:         |           |  page frame number |         |
               +-----------+--------------------+         |
             check access  |                    |         |
                           |                    |         |
                           |29                 9|8        V 0
                           +--------------------+-----------+
physical address of data:  |  page frame number |   byte    |
                           +--------------------+-----------+
```

Figure 11-2 rtVAX Process-Space Address Translation

11.2.5 Vector Processor

A processor may implement or omit the vector processor only as a whole. That is, if any vector facility is implemented (omitted), all vector facilities are implemented (omitted). If a processor omits the vector processor, execution of a vector instruction results in a reserved-instruction fault.

11.2.6 Address Space Number Register (ASN)

A processor may choose to identify the process address space provided to a process by an address space number (ASN), which the processor maintains in the ASN register. If this option is implemented, the following must also be included:

1. Address Space Number (ASN) IPR and its image in the PCB;

2. Previous CPU field (PRVCPU) in the PCB;

3. CPUID IPR;

4. TB invalidate virtual address space number (TBIASN) IPR;

5. A mechanism to associate address space numbers with per-process TB state;

6. modification of SVPCTX operation to write CPUID IPR to PCB<PRVCPU>;

7. modification of LDPCTX operation to support ASN IPR.

If a processor implements this option and also supports Virtual Machines then it must associate the value of SBR with system TB state and include the TB invalidate system (TBISYS) IPR.

11.2.7 UNIBUS interconnect

If a UNIBUS interconnect is supported on a processor implementation, then all the following requirements for its support must be included.

- Interrupt levels 14 through 17 (hex) corresponding to the UNIBUS levels BR4 through BR7. (See sections 5, 5.3.2, and 5.6.2.)

- UNIBUS interrupt vectors in SCB pages. (See section 5.6.2.)

- One or more areas of "UNIBUS I/O space", each 2**18 bytes in length, that map to UNIBUS addresses. (See sections 7.5 and B.1.1.)

- Certain I/O-bus adapters need separate TB entries for each UNIBUS memory page they can map, and they pre-load all needed TB entries.

- Some byte and word references of read-modify-write type in UNIBUS I/O spaces (including ADAWI) interlock correctly using the DATIP and DATO(B) functions. (See sections 7.5.2, 7.1.4, 7.5.2, and 9.13.)

11.3 INSTRUCTION EMULATION

VAX processors and their operating systems cooperate to support emulation of those instructions that are omitted from the processor's instruction set. Programs running under the operating system can make use of these instructions as though they were supported directly by the processor.

The mechanism for emulating an omitted instruction depends on the instruction type. Emulation of instructions in the following groups is done entirely by software. Upon execution, a reserved-instruction fault is taken.

- extended-accuracy group
- compatibility mode group
- floating-point instructions in the emulated-only group: ACB{F,D,G,H}, EMOD{F,D,G,H}, POLY{F,D,G,H}
- vector instructions (optional)

Emulation of string instructions in the following groups is assisted by the processor through the instruction-emulation exception

- packed-decimal-string group
- string instructions in the emulated-only group: MATCHC, MOVTC, MOVTUC, CRC, EDITPC

The process of emulating an omitted string instruction consists of the following steps:

1. The processor reads the instruction opcode and finds that this is an omitted instruction. The processor saves the opcode.
2. The processor evaluates the operand specifiers in order of instruction stream occurrence. The processor saves the operand address for each operand of write-access type or address type, and it reads and saves the operand itself for operands of read-access type. \Operands of other types do not occur in any of the emulated instructions.\
3. The processor initiates an instruction-emulation trap. The current stack is probed. If the probe faults, the instruction faults, else, an emulation trap frame is pushed onto the stack. The opcode and operands (or their addresses) are part of the trap frame. Unlike many exceptions, instruction-emulation trap does not cause the processor to enter kernel mode. The exception handler runs in the same mode as the trapped instruction, and the trap frame is pushed onto the current stack.

4. The emulation-exception handler in the operating system examines the opcode of the trapped instruction and dispatches to the appropriate emulation routine.

5. The instruction-emulation routine reads and writes the instruction operands, as appropriate to the instruction being emulated. The operands need not be probed, since the emulation handler is running in the same mode as the emulated instruction.

6. The instruction-emulation routine sets the condition codes in the PSL on the stack, pops the emulation trap frame (except for the new PC and PSL) from the stack, and returns with REI.

7. Emulation is now complete, and the instruction following the emulated instruction begins execution.

If, during the emulation of an instruction, an exception such as access violation occurs, the emulation code must gain control, save state in the registers just as the emulated instruction would, set FPD in the saved PSL, and reflect the exception to the user's current exception handler. If the conditions causing the exception are corrected and the exception was a fault, the instruction can be restarted. In this case, PSL<FPD> will be set when instruction execution begins. Emulation consists of the following steps:

1. The processor reads the opcode and finds that this is an omitted instruction and that PSL<FPD> is set.

2. The processor initiates a suspended-emulation fault. The current stack is probed. If the probe faults, then the instruction faults, else the processor pushes PC and PSL onto the stack.

3. The emulation-exception handler rebuilds the intermediate state of the instruction, using the information saved in the general registers at the time the emulated instruction was faulted.

4. The emulation handler resumes emulation of the instruction, as in steps 5 through 7 in the previous list above.

Emulation software runs in the mode of the emulated instruction and uses the same stack. Emulation software may allocate and use up to five pages of stack space for temporary storage. The contents of this area are UNPREDICTABLE after execution of an emulated instruction. If an emulated instruction addresses part of this area as an operand without first allocating it, or if an emulated instruction uses SP as an operand, the results of the instruction are UNPREDICTABLE. That is, the instructions DIVF3 R1, R2, -50(SP) and DIVF3 R1, R2, SP produce UNPREDICTABLE results. The instruction DIVF3 R1, R2, -(SP) allocates the area on top of the stack before using it and is legal.

11.3.1 Instruction-Emulation Exceptions

When a processor executes a string instruction that is omitted from its instruction set, an emulation exception results. An emulation exception occurs through one of two SCB vectors, depending on whether or not PSL<FPD> is set at the beginning of the instruction. If PSL<FPD> is clear, an instruction-emulation trap occurs through the SCB vector at offset C8 (hex), and an instruction-emulation trap frame is pushed onto the stack. The PC pushed points to the instruction following the omitted instruction. If PSL<FPD> is set, a suspended-emulation fault occurs through the SCB vector at offset CC (hex), and PC and PSL are pushed onto the stack. The PC pushed points to the faulted instruction.

In either case, if PSL<T> is set at the time of the trap, PSL<TP> is set in the PSL pushed onto the stack. All other bits in the pushed PSL are unchanged. If PSL<FPD> was set, it is set in the saved PSL.

The new PSL has <TP,FPD,IV,DV,FU,T> clear. All other fields are unchanged, including PSL<CUR_MOD,PRV_MOD,IS,IPL>. That is, the emulation-exception handler runs in the mode of the emulated instruction, on the same stack, and at the same IPL. The exception parameters are pushed onto the current stack. (If the current stack cannot be written, the processor takes a memory management fault rather than an emulation exception.)

If either emulation-exception vector has bits <1:0> set to 1 (indicating that the exception is to be taken on the interrupt stack), the operation of the processor is UNDEFINED.

The emulation-exception stack frame is shown in Figure 11-3 and includes the following:

- Opcode -- contains the opcode of the trapped instruction.

- Old PC -- contains the address of the trapped instruction.

- Specifiers 1 through 8 -- contain the addresses of corresponding instruction operands or contain the operands themselves. For each operand of the trapped instruction, if the operand is of read access type (.rx), the parameter contains the operand value; if the operand is write access type (.wx) or address type (.ax), the parameter contains the operand address. For read-type operands of byte size, bits <31:8> of the longword are UNPREDICTABLE. For read-type operands of word size, bits <31:16> are UNPREDICTABLE. When an operand is in a register, the register is denoted by a reserved system space address corresponding to the one's complement of the register number. The parameter corresponding to an instruction operand that does not exist is UNPREDICTABLE. For example, if the trapped instruction has four operands, the parameters for specifiers 5 through 8 are UNPREDICTABLE.

- New PC -- contains the address of the instruction following the trapped instruction.

- Saved PSL -- contains the PSL at the time of the trap. If PSL<T> was set at the beginning of the instruction, saved PSL<TP> is set. When the frame is created as a result of an instruction-emulation trap from a virtual machine, the saved PSL is the PSL of the virtual machine.

```
     31                                                            0
    +-----------------------------------------------------------------+
    |                             opcode                              | :(SP)
    +-----------------------------------------------------------------+
    |                             old PC                              |
    +-----------------------------------------------------------------+
    |                          specifier #1                           |
    +-----------------------------------------------------------------+
    |                          specifier #2                           |
    +-----------------------------------------------------------------+
    |                          specifier #3                           |
    +-----------------------------------------------------------------+
    |                          specifier #4                           |
    +-----------------------------------------------------------------+
    |                          specifier #5                           |
    +-----------------------------------------------------------------+
    |                          specifier #6                           |
    +-----------------------------------------------------------------+
    |                          specifier #7                           |
    +-----------------------------------------------------------------+
    |                          specifier #8                           |
    +-----------------------------------------------------------------+
    |                             new PC                              |
    +-----------------------------------------------------------------+
|   |                    saved PSL (or PSL of VM)                     |
    +-----------------------------------------------------------------+
```

Figure 11-3 Instruction-Emulation Trap Frame

Change History:

Revision J. Tom Eggers, Rich Brunner, December 1989
- o Vector instructions are optional to implement as a group. Vector instructions can be optionally emulated by software.
- o Make the UNIBUS interconnect an option.
- o Added ASN/Process-ID option.
- o Make it clear that a saved PSL may be the PSL of a VM.

Revision H. Tim Leonard, May 1987.
- o Make it clear that emulated-only instructions can be implemented one-by-one, they're not implemented or omitted only as a group.
- o Support virtual machines.
- o Revise the section names and ordering.
- o Minor edits for physical address extension.

Revision F. Cheryl Wiecek & Al Thomas, November 1986.
- o Rewrite first four sections to account for the new implementation rules.
- o Rename subset emulation to instruction emulation.

Revision E. Al Thomas, September 1986.
- o Clarify probe of current stack during instruction-emulation exception.
- o Add description of rtVAX memory management.
- o Add description of EMOD/POLY instruction-group subset.

Revision D. Tim Leonard, March 1985.
- o For read-type operands of byte or word size, upper bytes of the specifier longword in the emulation trap frame are UNPREDICTABLE.
- o On suspended-emulation fault, the pushed PC points to the emulated instruction.
- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Make it clear that operating systems may emulate all of the omitted instructions, and that they use the current stack for temporary storage.
- o Rename to Chapter 12.
- o Remove the term "MicroVAX subset" and replace it with more clearly defined terms.
- o Rename emulation exceptions.
- o Specify what happens when an emulation exception occurs and the stack faults.
- o Emulation exception vector low bits must be zero.
- o Move the description of the implemented subsets to Chapter 12.
- o Delete section describing operating system futures.
- o Add a list of the kernel instruction set.

Revision 6, new subsetting rules. Dileep Bhandarkar, 15 November 1983.

- o Update subsetting rules to reflect current processors.
- o Add MicroVAX subset.

Revision 5, add G_floating and H_floating. Dileep Bhandarkar, 6 September 1978.
Revision 4. Peter Conklin, 28 February 1977.

- o EDITPC ECO.
- o Packed decimal ECO.

Revision 3, ECOs 12 through 18 and results of April Task Force review. Peter Conklin, 27 May 1976.

- o Add MOVF, MOVD to subset.
- o Add CVT{B,W,L}{F,D} to subset.
- o Add ACBF, ACBD to subset.
- o Remove CHOPF, CHOPD, POLYF, POLYD.
- o Add CRC as a class.
- o Make EDITN a separate class.
- o Remove MOVU, MULN4, DIVN4, CVTLU, CVTPU, ASHU.
- o Change names to MOVN, MULN, DIVN, CVTLN, CVTPN, ASHN.
- o Add MOVTUC, MATCHC to character class.
- o Clarify subsetting a class.
- o Remove address break.
- o Document rules on future additions.
- o Note that CRC, MOVTUC, MATCHC are not currently committed for First Machine.
- o Add POLYF, POLYD.
- o Make MULN, DIVN a separate class.
- o Add SKPC.

Revision 2, made names agree with ECO 5. Tom Hastings, 11 March 1976.
Revision 1, results of pruning meeting. Tom Hastings, 18 December 1975.

CHAPTER 12

VIRTUAL MACHINES

A virtual machine is the software implementation of a computer architecture. A "virtual VAX" is a virtual machine that implements the VAX architecture. A "real VAX" is the hardware and microcode that implements the VAX architecture. The VAX architecture includes hardware and microcode features to support creation and execution of virtual VAXes.

The term "processor" refers to the "real VAX", and the term "VVAX" refers to the "virtual VAX".. Figure 12-1 illustrates the implementation of virtual VAXes by software running on a real VAX.

Virtual-VAX support is an optional part of the VAX architecture. When the option is implemented, several features, described in this chapter, are added to the architecture, and other basic features, described in other chapters, are modified. Processors that implement virtual-VAX support include:

1. The virtual-machine bit in the processor PSL register, described in Chapter 1

2. The virtual-machine PSL register, VMPSL, described in this chapter

3. Modifications to the instructions HALT, CHMx, LDPCTX, SVPCTX, MTPR, MFPR, MOVPSL, PROBEx, and REI

4. Instructions used only for virtual machines -- WAIT, PROBEVMW, and PROBEVMR -- described in this chapter

5. An exception used only for virtual machines, VM-emulation trap, described in this chapter

6. The modify-fault option, described in Chapter 5

7. Modifications to the logic for initiating an exception or interrupt, described in Chapter 5

A pure software implementation of a virtual VAX is one possible implementation strategy. In such an implementation, software running on a real VAX completely emulates all features of a VAX, such as general registers and a PSL. Software then "interprets" the virtual machine's instruction stream much as one could interpret PDP-8 instructions using VAX machine language. The drawback of such a scheme is that software emulation is very slow.

A more complicated but much faster technique relies on the fact the virtual VAX is running on a real VAX: most instructions can be executed by the real VAX hardware on behalf of the virtual machine. This scheme forces software running on the real VAX to keep a separation between the virtual context, or "state of the virtual VAX", and the real context, or "state of the real VAX."

This chapter describes the VAX architecture support for the faster technique. At certain times, the processor executes code on behalf of a virtual VAX. At other times, the processor executes code for the native-mode operating system, or for user code in a non-virtual machine environment. When the processor executes code for a virtual VAX, it is said to be "executing a virtual machine" or "running in VM mode". When the processor executes other code, it is said to be "running a real machine".

Many terms that are clear in other contexts become ambiguous or confusing when discussing virtual machines. Partly, this is because there are several machine interfaces to discuss, several operating systems, and so on. More often, though, confusion arises when the meaning of terms changes with a change in the viewpoint of the discussion. For example, from the viewpoint of a program running in a virtual machine, the phrase "the current PSL" is unambiguous. From the viewpoint of a program running in a real machine, that same piece of state is called "the virtual machine's PSL," and the phrase "the current PSL" now refers to the PSL of the real machine. The term is unambiguous, but its meaning depends on its context. To avoid confusion, we must always make the context clear.

The context assumed in this chapter is that of the real machine and the operating system running on the real machine. We distinguish the real machine's state from that of the virtual machine by using the prefixes "RM-" and "VM-".

```
+---------------+          +---------------+---------------+
|               |          |               |               |
| user programs |          | user programs | user programs |
|               |          |               |               |
|~~~~~~~~~~~~~~~|          |~~~~~~~~~~~~~~~|~~~~~~~~~~~~~~~|
|  OS interface |          |OS #1 interface|OS #2 interface|
|               |          |               |               |
|               |          |     OS #1     |     OS #2     |
|               |          |               |               |
|               |          |\_/\_/\_/\_/\_/|\_/\_/\_/\_/\_/|
|      OS       |          | virtual VAX #1 virtual VAX #2 |
|               |          |    interface     interface    |
|               |          |                               |
|               |          | virtual machine implementation|
|               |          |                               |
|\_/\_/\_/\_/\_/|          |\_/\_/\_/\_/\_/\_/\_/\_/\_/\_/\|
| VAX interface |          |       real VAX interface      |
|               |          |                               |
|     VAX       |          |              VAX              |
|               |          |                               |
+---------------+          +---------------+---------------+
```

Real VAX and
Operating System

Virtual VAXes and Operating Systems

Figure 12-1 Real VAXes and Virtual VAXes

```
 3       2 2 2 2 2 2 2 2       1 1                             0
 1       7 6 5 4 3 2 1 0       6 5                             
+---------+-+---+---+-+---------+-------------------------------+
|         |I|CUR|PRV|M|         |                               |
|   MBZ   |S|MOD|MOD|B|   IPL   |              MBZ              |
|         | |   |   |Z|         |                               |
+---------+-+---+---+-+---------+-------------------------------+
```

Figure 12-2 Virtual-Machine PSL Register (VMPSL)

Table 12-1: Fields of the Virtual Machine's PSL

Name	Mnemonic	Location
compatibility mode	CM	PSL<CM>
trace pending	TP	PSL<TP>
virtual machine mode	VM	value is always 0
first part done	FPD	PSL<FPD>
interrupt stack	IS	VMPSL<IS>
current mode	CUR_MOD	VMPSL<CUR_MOD>
previous mode	PRV_MOD	VMPSL<PRV_MOD>
interrupt priority level	IPL	VMPSL<IPL>
decimal overflow enable	DV	PSL<DV>
floating underflow enable	FU	PSL<FU>
integer overflow enable	IV	PSL<IV>
trace enable	T	PSL<T>
negative	N	PSL<N>
zero	Z	PSL<Z>
overflow	V	PSL<V>
carry	C	PSL<C>

12.1 VIRTUAL-MACHINE STATE

The Virtual-Machine Processor Status Longword Register (VMPSL), shown in Figure 12-2, is a register that exists only on processors that support virtual machines.

Most of a virtual machine's state is maintained by software. Part of the currently executing VM's PSL is kept in internal processor register VMPSL, and is used by processor instructions.

When PSL<VM> contains a 0, VMPSL is a read/write register with no effect. This means that values may be written to the register and it will hold them, and when the register is read, it will return the last value written to it. The register will have no other effects on the processor or execution flows of instructions. Values written to VMPSL must be properly saved because the method of starting a virtual machine involves first writing a value to VMPSL and then executing an REI instruction to enter the VM. This is the only way to initialize the register. \An implementation of VMPSL need hold only the non-MBZ fields.\

When PSL<VM> contains a 1, VMPSL contains those fields of the PSL of the virtual machine (for which the processor is executing code) required to define the state of that virtual machine, and the processor is executing that virtual machine. The relevant fields in VMPSL are IS, CUR_MOD, PRV_MOD, and IPL. The values of the remaining fields of VM's PSL are either found in the corresponding fields of PSL or are constant.

It is important to stress the difference between the VMPSL register and the virtual machine's PSL. The virtual machine's PSL is an abstraction that represents the state of the virtual machine. The virtual machine's PSL is not a processor register, and its value is not found in any one location. VMPSL is a processor register that contains fields of a virtual machine's PSL that are needed by hardware or microcode and are not otherwise calculable. Table 12-1 describes the fields of a virtual machine's PSL.

Processor initialization leaves the value of VMPSL UNPREDICTABLE.

When the processor is executing a virtual machine, the value of PSL is constrained by the value of VMPSL. Specifically,

```
        PSL<CUR_MOD> is equal to MAXU( VMPSL<CUR_MOD>, 1 );
        PSL<PRV_MOD> is equal to MAXU( VMPSL<PRV_MOD>, 1 );
        PSL<IPL> is zero;  and
        PSL<IS> is zero.
```

If these constraints are met on entry into a virtual machine (that is, when REI loads a PSL with VM set), the constraints will be preserved until the processor stops executing the virtual machine. If any of these constraints is not met, operation of the processor is UNDEFINED.

Among other things, the constraints imply that a processor executing a virtual machine is not in kernel mode, and is not executing on the kernel stack or on the interrupt stack.

As can be seen in Table 12-1, the VM field of a virtual machine's PSL is always zero. This does not preclude self-virtualization. (Self-virtualization means the ability to implement a virtual machine on top of another virtual machine.) The reason that the VM field is always zero is that the VMPSL register corresponds to selected portions of the PSL of whatever level of virtual machine is currently running. That virtual machine is, by definition, being interpreted by the processor, and its VM bit is clear. Software is responsible for keeping and maintaining the PSLs of all intermediate levels of virtual machines. VM will be set in the PSLs of all intermediate-level VMs.

12.2 INSTRUCTIONS TO SUPPORT VIRTUAL MACHINES

Three instructions are added to the VAX repertoire in order to support virtual machines. They are WAIT, PROBEVMR, and PROBEVMW.

The WAIT instruction causes a privileged-instruction fault. It is provided to allow a way for a virtual machine to signal to the real-machine operating system (RMOS) that idle time is available. The RMOS may use this information to decide to schedule some other virtual machine.

PROBEVMR and PROBEVMW are special kinds of probe operations, provided as performance enhancements. In implementing virtual machines, the RMOS must probe the virtual machine's memory on behalf of the virtual machine. This is a fairly frequent operation and is performance sensitive. These new instructions provide exactly the probe operations needed by the RMOS.

WAIT Wait for Interrupt

Purpose: To signal idle processor time in a virtual machine

Format:

```
        opcode
```

Operation:

```
        {privileged instruction fault};
```

Condition codes:

```
        N <- 0; !condition codes cleared after reserved
        Z <- 0; ! instruction fault
        V <- 0;
        C <- 0;
```

Exceptions:

privileged instruction

Opcodes:

02FD	WAIT	Wait for Interrupt

Description:

The hardware treats this opcode as a privileged instruction and faults.

The real machine operating system (RMOS) handles the privileged instruction fault. If the processor was not executing a virtual machine, the RMOS treats the fault as it would any other privileged instruction in the RMOS. If the processor was executing a virtual machine and the virtual machine was not in kernel mode, the RMOS passes a privileged instruction fault to the virtual machine. If the processor was executing a virtual machine and the virtual machine was in kernel mode, the RMOS places the virtual machine in a wait state. The virtual machine will leave the wait state if it receives an interrupt whose priority is greater than that in VMPSL<IPL>, the virtual machine's interrupt priority level.

Notes:

1. The implementation of this instruction is different for real machines than it is for the virtual VAX processor.

2. The PSL in the exception frame is the PSL at the beginning of the instruction, including the condition codes.

3. In the virtual VAX processor, the virtual machine enters a wait state upon executing a WAIT instruction. The virtual

machine may leave the wait state at any time, at the discretion of the RMOS. The virtual machine must leave the wait state if the virtual machine receives an interrupt whose priority is greater than that in VMPSL <IPL>. Software in the virtual machine must not assume that an interrupt has been received when the virtual machine continues execution after executing a WAIT instruction.

PROBEVMx Probe Virtual Machine Accessibility

Purpose: Verify that a byte can be accessed by a virtual machine

Format:

```
        opcode  mode.rl, base.ab
```

Operation:

```
        probe_mode <- mode;
        if PSL<VM> EQLU 1 AND VMPSL<CUR_MOD> EQLU 0 then
                {initiate VM-emulation trap};
                ! Mode and base address are pushed
                !  onto the exception frame.
        if PSL<CUR_MOD> NEQ 0 then
                {privileged-instruction fault};

        probe_mode <- MAXU ( 1, probe_mode<1:0> );
        condition codes <- {accessibility of base
                                    using probe_mode};
```

Condition Codes:

```
        N <- {access control violation};
        Z <- {reference allowed};
        V <- {translation not valid};
        C <- {page not previously modified and PROBEVMW};
```

Exceptions:

privileged instruction fault
VM emulation

Opcodes:

9AFD	PROBEVMR	Probe Virtual Machine Read Accessibility
9BFD	PROBEVMW	Probe Virtual Machine Write Accessibility

Description:

The PROBEVMx instructions check the read or write accessibility of the byte specified by the base address. The protection is checked against the larger (and therefore less privileged) of the mode specified in bits<1:0> of the mode operand and executive mode (mode 1).

Notes:

1. Exactly one of the condition codes is 1 and all others are 0 after the execution of one of the PROBEVMx instructions.

2. The accessibility conditions are tested in the following order: accessibility, validity, and previous modification.

The meaning of the condition codes is as follows: If PSL<N> is set, then the byte is inaccessible. If PSL<V> is set, then the byte is accessible but not valid. If PSL<C> is set, then the instruction is PROBEVMW, the byte is accessible and valid, but the page was not previously modified. Otherwise, PSL<Z> is set.

3. PSL<C> is set if and only if the instruction was PROBEVMW, the target byte is both accessible and valid, and the page containing the target byte was not previously modified (as defined by a clear PTE<M> in the mapping PTE). Since the page is valid and accessible, it will remain so until the virtual machine is allowed to execute.

4. If PSL<C> is set, then PTE<M> was clear before the PROBEVMW instruction started. On the VAX 8800 family, the processor sets PTE<M> in the mapping PTE.

5. A length violation gives a status of "access violation".

6. 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.

7. The PROBEVMx instructions differ from the PROBEx instructions in the following ways:

   o PROBEVMx is privileged; it can only be executed from kernel mode.

   o PROBEVMx does not take a length operand; it probes only one byte.

   o The probe_mode is maximized with 1 rather than with PSL<CUR_MOD>.

   o Validity of the page and (PROBEVMW only) whether PTE<M> is clear in the mapping PTE are tested, as well as the accessibility of the target byte.

   o The sense of PSL<Z> is reversed from that of the PROBEx instructions.

   o PROBEVMW never takes a modify fault; PROBEW may.

8. The PROBEVMx instructions are intended for use by RMOS software. They allow the RMOS to determine whether a VM may access a specific location. In this context, accessibility of the probe mode is checked against the protection codes in the page tables that are known to the real processor.

9. The two PROBEVMx instructions belong to the virtual-machine instruction group, which is optional to implement. For more

detail, refer to Chapter 11.

Example:

```
        MFPR    #PR$_VMPSL, R0            ; Read the virtual
                                          ;  machine's PSL.
        ROTL    #-24, R0, R0              ; Get VMPSL<CUR_MOD> into
                                          ;  R0<1:0>.

        MOVL    VMDATA$GL_VMKSP, R1       ; Get the VM's kernel
                                          ;  stack pointer.
        PROBEVMR R0, (R1)                 ; Can first byte of first
                                          ;  arg be read?
        BNEQ    PROBE_FAILURE             ; If not, go handle
                                          ;  problem.

        PROBEVMR R0, 3(R1)                ; Can last byte of first
                                          ;  arg be read?
        BNEQ    PROBE_FAILURE             ; If not, go handle
                                          ;  problem.

        MOVL    8(AP), R1                 ; Get address of second
                                          ;  argument.
        PROBEVMW R0, (R1)                 ; Can first byte of arg 2
                                          ;  be written?
        BNEQ    PROBE_FAILURE             ; If not, go handle
                                          ;  problem.

        PROBEVMW R0, 7(R1)                ; Can last byte be
                                          ;  written?
        BEQL    PROBING_DONE              ; If so, skip error
                                          ;  handling.

PROBE_FAILURE:
        BLSS    ACCESS_VIOLATION          ; Return access violation
                                          ;  fault to the virtual
                                          ;  machine.
        BVS     TRANSLATION_NOT_VALID     ; Check for shadow page
                                          ;  table miss, or VM
                                          ;  translation not valid.
        BCS     SET_PTE_M                 ; Set the modified bit in
                                          ;  the VM's PTE, then
                                          ;  continue with other
                                          ;  probes.

PROBING_DONE:
```

12.3 EMULATION OF VIRTUAL-MACHINE INSTRUCTIONS

Most instructions executed by a virtual machine are executed directly by the processor. Under some circumstances, the processor cannot execute an instruction directly but must be assisted by the real-machine operating system (the RMOS). Whenever the processor cannot directly execute a virtual machine's instruction, the processor initiates an exception, transferring control to the RMOS. The RMOS can simulate the appropriate operation in the virtual machine and can execute REI to cause the processor to resume execution of the virtual machine.

When a virtual machine executes a privileged instruction in kernel mode, or when a virtual machine executes a change-mode or REI instruction, or in some cases when a virtual machine executes a PROBEx instruction, the processor initiates a VM-emulation trap to transfer control to the RMOS. The instructions that can cause a VM-emulation trap are HALT, PROBEx, REI, CHMx, LDPCTX, SVPCTX, MTPR, MFPR, PROBEVMW, and PROBEVMR.

In initiating a VM-emulation trap, the processor performs the following steps:

1. If PSL<TP> is set, a trace fault occurs.

2. Evaluate the operand specifiers in order of instruction stream occurrence. For modify-type, write-type, and address-type operands, save the address of the operand. The address of a register is denoted by a reserved, system-space address corresponding to the one's complement of the register number. (For example, the address of R5 is FFFFFFFA, hex.) The operand need not be probed because only its address is being saved; RMOS software must probe the operand before accessing it. For read-type operands, probe the operand, read it, and save its value. If access is not allowed, take an access-control-violation fault; if the page is not valid, take a translation-not-valid fault. Note that a modify fault will not be taken on an emulated instruction or its operands. This is because no write-type or modify-type operand is written. Instead, the operand's address is saved on the stack. RMOS software, as part of emulating the instruction, will set any PTE<M> bits that need to be set.

3. The information saved on the exception stack frame includes the PSL of the RM at the time of the exception, the PC of the trapped instruction, the PC of the instruction that follows the trapped instruction, up to three operand specifiers, and the opcode of the trapped instruction. See Figure 12-3 and Table 12-2.

4. Initiate an exception in kernel mode through the VM-emulation trap vector using 38 (hex) as the SCB offset. The TP bit in the saved PSL is set if PSL<T> was set.

```
 3
 1                                                        0
+--------------------------------------------------------+
|                         opcode                         | :(SP)
+--------------------------------------------------------+
|                PC of trapped instruction               | 4
+--------------------------------------------------------+
|                  instruction operand #1                | 8
+--------------------------------------------------------+
|                  instruction operand #2                | 12
+--------------------------------------------------------+
|                  instruction operand #3                | 16
+--------------------------------------------------------+
|     PC of instruction following trapped instruction    | 20
+--------------------------------------------------------+
|                 saved PSL (of the RM)                  | 24
+--------------------------------------------------------+
```

Figure 12-3 VM-Emulation-Trap Stack Frame

Table 12-2: Contents of the VM-Emulation-Trap Stack Frame

Offset (decimal)	Contents
00	The zero-extended opcode of the trapped instruction.
04	The PC of the trapped instruction.
08 - 16	Instruction operands 1 through 3. For operands of read access-type, the associated longword contains the operand value. For operands of modify, write, or address access-type, the associated longword contains the address of the operand. The address of a register is denoted by a reserved, system-space address corresponding to the one's complement of the register number. Any longwords corresponding to nonexistent operands contain UNPREDICTABLE contents. The high word of longwords containing word-size operands and the high three bytes of longwords containing byte-size operands are UNPREDICTABLE.
20	PC of the instruction following the trapped instruction.
24	PSL (of the RM) at the beginning of the execution of the trapped instruction, including condition codes.

12.3.1 Emulation of Instruction-Emulation Traps

Some VAX processors do not implement the full VAX instruction set in hardware and microcode. These processors take an instruction-emulation trap when they encounter an instruction that is to be implemented in software. Virtualization of instruction-emulation traps is slightly complicated because the real hardware does not know the address of a virtual machine's SCB. Therefore, the real machine must trap to RMOS software to properly deliver the instruction-emulation trap.

When the processor is in VM mode, the process of emulating an omitted instruction which causes an instruction-emulation trap consists of the following steps:

1. If PSL<TP> is set, a trace fault occurs.

2. The processor reads the instruction opcode, and finds that this is an omitted instruction. The processor saves the opcode.

3. The processor evaluates the operand specifiers in order of instruction stream occurrence. The processor saves the operand address for each operand of write-access type or address type, and it reads and saves the operand itself for operands of read-access type. (Operands of modify, field, and branch types, and of size longer than longword, do not occur for instructions that might cause either an instruction-emulation or suspended-emulation exception.)

4. The current stack is probed. If the probe faults, the instruction faults, else an instruction-emulation trap frame (see Figure 11-3) is pushed onto the current stack. The PSL saved in the stack frame is the PSL of the VM (and not the PSL of the RM). The opcode and operands (or their addresses) are part of the trap frame.

5. The processor initiates a VM-emulation trap in kernel mode. The opcode field of the VM-emulation trap stack frame contains the opcode of the omitted instruction. The contents of the instruction operand fields are UNPREDICTABLE. The other fields in the VM-emulation-trap frame contain valid information.

6. The RMOS VM-emulation exception handler examines the opcode of the trapped instruction, determines that the trap was due to an omitted instruction, reads the SCB of the virtual machine to obtain the instruction-emulation-trap vector, and causes an instruction-emulation exception in the VM, in the mode from which the omitted instruction was executed.

7. Processing continues with step 4 of the instruction emulation process, described in section 11.3.

12.3.2 Emulation of Suspended-Emulation Faults

Similar considerations apply when dealing with a suspended-emulation fault. When the processor is in VM mode, the process of emulation consists of the following steps:

1. The processor reads the opcode, and finds that this is an omitted instruction, and that PSL<FPD> is set.

2. The current stack is probed. If the probe faults, then the instruction faults, else the processor pushes PC and the PSL of the VM onto the current stack.

3. The processor initiates a VM-emulation trap in kernel mode and builds a VM-emulation-trap stack frame. The opcode of the omitted instruction is placed in the opcode field of the stack frame. The PC of the trapped instruction is also placed in the corresponding field of the stack frame. The fields in the stack frame for the instruction operands and the PC of the instruction following the trapped instruction are UNPREDICTABLE since these values are not needed.

4. The RMOS VM-emulation exception handler examines the opcode of the trapped instruction, determines that the trap was due to an omitted instruction, determines that PSL<FPD> in the VM-emulation stack frame is set, reads the SCB of the virtual machine to obtain the suspended-emulation fault vector, and causes a suspended-emulation exception in the VM, in the mode from which the omitted instruction was executed.

5. Processing continues with step 3 of the suspended-instruction emulation process, described in section 11.3.

12.4 RING COMPRESSION

The VAX Architecture was not originally designed to be virtualizable. To create virtual VAXes, solutions had to be found for problems posed by the original design of the VAX.

The most difficult part of virtualizing the VAX is the access-mode scheme. The design of a secure system requires that the most privileged mode, kernel mode, be reserved for the RMOS. In order to provide the virtual machine with four access modes, either a fifth access mode needs to be invented, or some way must be found for four virtual machine access modes to fit into three real-machine access modes. In the interests of minimizing changes to the VAX architecture, a method was found to fit four access modes into three. This method is called "ring compression."

```
+-----------------+      +-----------------+
|                 |      |                 |
|    user mode    |      |    user mode    |
|                 |      |                 |
+-----------------+      +-----------------+
|                 |      |                 |
| supervisor mode |      | supervisor mode |
|                 |      |                 |
+-----------------+      +-----------------+
|                 |      |                 |
|                 |      |  executive mode |
|                 |      |                 |
| executive mode  |      +.................+
|                 |      |                 |
|                 |      |   kernel mode   |
|                 |      |                 |
+-----------------+      +-----------------+
|                 |
|   kernel mode   |     Virtual-Machine Access Mode
|                 |
+-----------------+

    Real-Machine Access Mode
```

Figure 12-4 Use of Access Modes on Virtual VAX

Kernel mode on the real machine is reserved exclusively for use by the software that implements the virtual machines, the RMOS. User mode and supervisor mode in a virtual machine correspond to user mode and supervisor mode, respectively, on the real machine. Executive mode and kernel mode in a virtual machine are forced to share executive mode on the real machine, so that, in effect, there is no access mode boundary between them. The result is that code running in executive mode in a VM can read pages that are readable to kernel mode in that VM, even if the pages have protection codes of KR. Similarly, code running in executive mode in a VM can read and write pages that are writable to kernel mode in that VM, even if the pages have protection codes of KW.

The way that ring compression is accomplished is that the RMOS manipulates the protection fields in page tables. The virtual machine operating system (VMOS), as part of its memory management, creates a set of page tables. These are called the VM's page tables. The RMOS creates another set of page tables, called shadow page tables. The shadow page tables are the ones known to the real processor through the memory management registers. In creating the shadow page tables, the RMOS translates the protection codes from those set up by the VMOS to the appropriate codes that combine the VM's kernel and executive modes into the real machine's executive mode.

12.5 DIFFERENCES BETWEEN VIRTUAL VAXES AND REAL VAXES

Within the virtual machine, pages marked as readable only in kernel mode are also readable in executive mode; pages marked as writable only in kernel mode are also writable in executive mode. In effect, kernel mode and executive mode have been combined into a single access mode for page protection.

Within the virtual machine, the PROBER and PROBEW instructions are required to report whether addresses are actually accessible. In other words, as far as the PROBEx instructions are concerned, kernel mode and executive mode have been combined into a single access mode. In contrast, in the real machine, kernel mode and executive mode are distinct access modes, and the PROBEx instructions must be able to distinguish them.

The WAIT instruction may be executed by a virtual machine to signal processor idle time. When executed by a virtual machine, WAIT has no effect except on timing. When executed by a real machine, WAIT causes a privileged-instruction fault.

Virtual VAXes have restrictions in how much system and process virtual memory they can use. The restriction on the usage of system virtual memory allows the RMOS to occupy addresses in the system space of each VM. It is desirable to have the RMOS occupy the same addresses in the system space of each virtual machine.

To allow the RMOS and the VMOS to coexist in system space, the VM is allowed the lower part of system space, and the RMOS owns the higher part. The RMOS will prevent the VMOS from extending its system virtual-address space beyond the start of the RMOS.

The restriction on the usage of process virtual memory is that no process running on any virtual machine may have a total process virtual memory (that is, the sum of the sizes of P0 space and P1 space) that exceeds some value. The reason for this restriction is that the RMOS is responsible for building a "shadow page table," which describes to the real hardware the addresses and protections of all pages belonging to the VM. Some upper bound for the size of process virtual memory must be chosen at system boot time so that the RMOS can allocate sufficient table space.

Finally, a virtual VAX can differ from a real VAX in any way that real VAXes are allowed to differ. In particular, the execution speed of instructions is different, the structure of I/O is different, and the virtual VAX supports its own implementation-dependent internal processor registers. See Appendix B, Implementation Dependencies, for a description of these differences.

12.6 COMPATIBILITY MODE

Implementation of compatibility mode depends on whether the physical processor underlying the RMOS supports compatibility mode. If the physical processor supports compatibility mode, then the VVAX will. If the physical processor does not, then the VVAX will not. The value of the VVAX type field of the SID internal processor register will indicate to VMOS software whether compatibility mode is supported on an individual processor.

Compatibility mode may be supported by the following technique. A compatibility-mode exception is received by RMOS software in real kernel mode. The RMOS then forwards the exception to the VMOS through the VMOS's SCB. No additional hardware support is needed.

12.7 TIME ON VIRTUAL MACHINES

When dealing with clocks on virtual machines, one directly confronts the issue of virtual time versus real time. Real time means the ordinary human notion of time, as measured by a wall clock. Virtual time is a complicated fiction.

On a physical VAX, the time is maintained in a memory location by the operating system. The interval timer is used to issue interrupts every ten milliseconds. On each timer interrupt, software adds ten milliseconds to the time value. In this way, software always has an approximately accurate estimate of the real time.

On a virtual VAX, the above scheme has drawbacks. To virtualize the interval timer, the real timer would issue interrupts to the RMOS, and the RMOS would issue "virtual timer" interrupts to the virtual machines. If the RMOS were to issue a virtual timer interrupt to each virtual machine on each (real) clock tick, then the overhead required to maintain the time is increased by a factor equal to the number of virtual machines running on the real machine. If, on the other hand, the RMOS were to issue a virtual timer interrupt only to the currently running VM, then the software time values maintained in each VM are updated less frequently than every ten milliseconds. This causes a virtual machine to maintain a "time value" that is considerably different from the real time. Virtual time is the estimate of the real time that is maintained by a VM.

An implementation of virtual machines should explicitly make some decision about how the time is to be communicated to virtual machines.

Change History:

Revision J. Rich Brunner, Tom Eggers, Tim Leonard, December 1989

- For the emulation of suspended-emulation faults, when the processor is building the VM-emulation-trap stack frame, the "PC of instruction following trapped instruction" field in the stack frame is UNPREDICTABLE.
- PSL of RM pushed on stack for emulation of virtual-machine instructions.
- PSL of VM pushed on stack for emulation of instruction-emulation traps.
- ECO 114 -- Change PROBEVMx mode operand to longword.
- Fix PSL of VM bug.

Revision H. Tim Leonard, Tom Eggers, May 1987.

- Initial version. Drew Mason and Tim Leonard.

CHAPTER 13

VAX VECTOR REGISTERS AND INSTRUCTIONS

13.1 INTRODUCTION

This chapter and the next describe an extension to the VAX architecture for integrated vector processing. This chapter describes the vector registers and vector instructions of the VAX Vector Architecture. Chapter 14 describes the execution model and exception reporting for the vector architecture.

Some of the VAX vector architecture departs from the traditional VAX scalar architecture, especially in the areas of UNPREDICTABLE results, vector processor exceptions, and instruction/memory synchronization.

\ There were several constraints imposed on the definition of the VAX vector architecture.

- o The features should be identical to those of PRISM vectors. This was necessary to leverage the compiler off the PRISM vectorizing FORTRAN compiler and to allow the use of common VLSI building blocks.

- o The architecture had to be implementable on the new ECL machines (Aquarius) as well as VLSI machines like Rigel. The impact on the base scalar machines was to be minimized so as not to adversely impact existing schedules for the scalar processors.

- o The impact on VMS was to be minimized.

The architecture described here reflects these constraints. The areas most affected were the execution model, exception handling, and opcode assignments. Without these constraints the architecture might have been cleaner aesthetically, but a lot less useful to the corporation. \

Implementation of the VAX vector architecture is optional. VAX processors that do implement the vector architecture do so as specified in these two chapters. Operating system software may emulate the vector architecture on processors that omit this feature.

\ Emulating vector instructions will yield a significantly slower execution time than using scalar instructions to do the same operation. Emulation is neither required nor forbidden by the architecture. \

On VAX processors that omit the vector architecture, vector instructions result in a reserved-instruction fault.

The vector architecture features include additional instructions, vector registers, and vector control registers.

All examples of vector instructions follow the assembler notation specified at the end of this chapter. The number and order of operands for the assembler notation is different from the instruction-stream format specified in the vector instruction descriptions.

13.2 REGISTERS

13.2.1 Vector Registers

There are 16 vector registers, V0 through V15. Each vector register contains 64 elements numbered 0 through 63. Each element is 64 bits wide. Figure 13-1 depicts a vector register.

A vector instruction that performs a register-to-register operation is defined as a "vector operate instruction". A vector operate instruction that reads or writes F_floating data, or integer data for shifts or integer arithmetic operations, reads bits <31:0> of each source element and writes bits <31:0> of each destination element. Bits <63:32> of the destination are UNPREDICTABLE for F_floating, integer arithmetic, and shift instructions.

Vector logical instructions read bits <31:0> of each source element and write the result into bits <31:0> of each destination element; bits <63:32> of the destination element receive bits <63:32> of the corresponding element of the Vb source operand.

For vector instructions that read longword data from memory into a vector register (VLDL and VGATHL), bits <63:32> of the destination elements are UNPREDICTABLE.

If the same vector register is used as both source and destination in a Gather Memory Data into Vector Register (VGATH) instruction, the result of the VGATH is UNPREDICTABLE.

For the IOTA vector instruction, bits <63:32> of the destination elements are UNPREDICTABLE.

13.2.2 Vector Control Registers

The 7-bit Vector Length Register (VLR) limits the highest vector element to be processed by a vector instruction. VLR is loaded prior to executing the vector instruction using a MTVP instruction. The value in VLR may range from 0 to 64. If the vector length is zero, no vector elements are processed. If a vector instruction is executed with vector length greater than 64, its results are UNPREDICTABLE. Elements beyond the vector length in the destination vector register are not modified. See Figure 13-2.

The Vector Mask Register (VMR) has 64 bits, each corresponding to an element of a vector register. Bit <0> corresponds to vector element 0. See Figure 13-3.

The 7-bit Vector Count Register (VCR) receives the length of the offset vector generated by the IOTA instruction. See Figure 13-4.

These registers are read and written by Move From/To Vector Processor (MFVP/MTVP) instructions.

13.2.3 Internal Processor Registers

The vector processor contains the following IPRs that can be accessed by the scalar processor using MTPR/MFPR instructions:

- Vector Processor Status Register (VPSR)
- Vector Arithmetic Exception Register (VAER)
- Vector Memory Activity Check (VMAC)
- Vector Translation Buffer Invalidate All (VTBIA)
- Vector State Address Register (VSAR)

The VPSR is shown in Figure 13-5, and described in Table 13-1. Table 13-2 shows the possible settings of VPSR<3:0> in the same MTPR instruction, and the resulting action for the vector processor. The state of the vector processor is determined by the encoding of Vector Processor Enabled (VPSR<VEN>) and Vector Processor Busy (VPSR<BSY>). The vector processor state for possible encodings is shown in Table 13-3. Note that because the vector and scalar processors can execute asynchronously, a VPSR state transition may not be seen immediately by the scalar processor. After performing an MTPR to VPSR, software must then issue an MFPR from VPSR to ensure that the new state of VPSR (and VAER if cleared by VPSR<RST>) will affect the execution of subsequently issued vector instructions. The MFPR in this case will not complete until the new state of the vector processor becomes visible to the scalar processor. If software does not issue the MFPR, then it is UNPREDICTABLE whether this synchronization between the new state of VPSR (and VAER) and subsequently issued vector instructions occurs.

The VAER is shown in Figure 13-6. This read-only register is used to record information regarding vector arithmetic exceptions. Table 13-4 shows the encoding for the exception condition types. The destination register mask field of VAER records which vector registers have received default results due to arithmetic exceptions. VAER<16+n> corresponds to vector register Vn, where n is between 0 and 15. For more information, refer to section 14.2.2, Vector Arithmetic Exceptions.

The VMAC register is shown in Figure 13-7. This read-only register is used to guarantee the completion of all prior vector memory accesses. For more information on this function of VMAC, refer to section 14.3.2.2. An MFPR from VMAC also ensures that all hardware errors encountered by previous vector memory instructions are reported before the MFPR completes. For more information on this function of VMAC, refer to section 14.5, Hardware Errors. The value returned by MFPR from VMAC is UNPREDICTABLE.

The VTBIA register is shown in Figure 13-8. This write-only register may be omitted in some implementations. If the vector processor contains its own translation buffer, moving zero into VTBIA using the MTPR instruction invalidates the entire vector translation buffer. For more information, refer to section 14.4, Memory Management.

The VSAR is shown in Figure 13-9. This read/write register contains a quadword-aligned virtual address of memory assigned by software for storing implementation-specific vector hardware state when the asynchronous method of handling memory management exceptions is implemented. The length of this memory area is implementation-specific. Software must guarantee that accessing the memory pointed to by the address does not result in a memory management exception. If the synchronous method of handling memory management exceptions is implemented, this register is omitted. For more information, refer to section 14.2.1, Vector Memory Management Exception Handling.

With the exception of VPSR (and VAER), an MTPR to any other writable vector IPR ensures that the new state of the IPR affects the execution of all subsequently issued vector instructions. Vector instructions issued before an MTPR to any writable vector IPR are unaffected by the new state of the IPR (and any implicitly changed vector IPR) except in one case: when the MTPR sets VPSR<RST> while VPSR<BSY> is set. (See Table 13-1 for more details.)

Except for the cases listed below, the operations of the scalar and vector processors are UNDEFINED after execution of MTPR to a read-only vector IPR, MTPR to a nonexistent vector IPR, MTPR of a non-zero value to a MBZ field, or MTPR of a reserved value to a vector IPR. The preferred implementation is to cause reserved-operand fault.

1. If an implementation supports an optional vector processor, but the vector processor is not installed, MTPR to VPSR has no effect.

2. If an implementation supports an optional vector processor, but either the vector processor is not installed, or the scalar/vector processor pair uses a common translation buffer, MTPR to VTBIA has no effect. \ Implementations should attempt to optimize this operation since software uses it frequently. \

In each of these cases, MTPR is implemented as a no-op.

Except for the cases listed below, the operations of the scalar and vector processors are UNDEFINED after execution of MFPR from a nonexistent vector IPR, or MFPR from a write-only vector IPR. The preferred implementation is to cause reserved-operand fault.

1. If an implementation supports an optional vector processor, but the vector processor is not installed, MFPR from VPSR returns zero. \ Implementations should attempt to optimize this operation since software uses it frequently. \

2. If an implementation supports an optional vector processor, but the vector processor is not installed, MFPR from VMAC has no effect. \ Implementations should attempt to optimize this operation since software uses it frequently. \

The IPR assignments for these registers are found in Table 13-5.

```
 6
 3                                                          0
+-------------------------------------------------------------+
|                          element 0                          | :Vn
+-------------------------------------------------------------+
|                              .                              |
|                              .                              |
|                              .                              |
+-------------------------------------------------------------+
|                          element 63                         |
+-------------------------------------------------------------+
```

Figure 13-1 Vector Register

```
 3
 1                                                 7 6          0
+---------------------------------------------------+-------------+
|                      SBZ/RAZ                      |   length    |
+---------------------------------------------------+-------------+
```

Figure 13-2 Vector Length Register (VLR)

```
 6
 3                                                           1 0
+-+-----------------------------------------------------------+-+-+
| |                                                           | | |
| |                        .    .    .                        | | |
| |                                                           | | |
+-+-----------------------------------------------------------+-+-+
```

Figure 13-3 Vector Mask Register (VMR)

```
 3
 1                                                 7 6          0
+---------------------------------------------------+-------------+
|                      SBZ/RAZ                      |    count    |
+---------------------------------------------------+-------------+
```

Figure 13-4 Vector Count Register (VCR)

```
 3 3         2 2 2
 1 0         5 4 3                          8 7 6 5 4 3 2 1 0
+-+---------+-+-+---------------------------+-+-+-+-+-+-+-+-+
|B|         |I|I|                           |A|P|M| |R|S|R|V|
|S|    0    |V|M|             0             |E|M|F|0|L|T|S|E|
|Y|         |O|P|                           |X|F| | |D|S|T|N|
+-+---------+-+-+---------------------------+-+-+-+-+-+-+-+-+
```

Figure 13-5 Vector Processor Status Register (VPSR)

Table 13-1: Description of the Vector Processor Status Register

Extent	Type	Description
<0>	R/W	Vector Processor Enabled (VEN). The vector processor is enabled by writing a one to this bit. Writing a zero disables the vector processor. If VPSR<VEN> is cleared by software while VPSR<BSY> is set, then once the new state of VPSR becomes synchronized with subsequent vector instructions, no more are sent and the vector processor completes execution of all pending instructions in its instruction queue. See section 14.2.3, Vector Processor Disabled, for more details.
<1>	W	Vector Processor State Reset (RST). Writing a one to this bit clears VPSR and VAER. If VPSR<RST> is set by software while VPSR<BSY> is set, the operation of the vector processor is UNDEFINED. This bit is RAZ.
<2>	W	Vector State Store (STS). Writing a one to this bit initiates storing of implementation-specific vector state information to memory using the address in VSAR for the asynchronous method of handling memory management exceptions. If the synchronous method is implemented, writes to VPSR<STS> are ignored. This bit is RAZ.
<3>	W	Vector State Reload (RLD). Writing a one to this bit initiates reloading of implementation-specific vector state information from memory using the address in VSAR for the asynchronous method of handling memory management exceptions. If the synchronous method is implemented, writes to VPSR<RLD> are ignored. This bit is RAZ.
<4>	R	0
<5>	R/W1C	Memory Fault (MF). This bit is set by the vector processor when there is a memory reference to be retried due to an asynchronous memory management exception. Writing a one to VPSR<MF> clears it. Writing a zero to VPSR<MF> has no effect. If the synchronous method of handling memory management exceptions is implemented, this bit is always zero.
<6>	R/W1C	Pending Memory Fault (PMF). This bit is set by the vector processor when an asynchronous memory management exception is pending. Writing a one to VPSR<PMF> clears it. Writing a zero to VPSR<PMF> has no effect. If the synchronous method of handling memory management exceptions is implemented, this bit is always zero.

Table 13-1: Description of the Vector Processor Status Register

Extent	Type	Description
<7>	R/W1C	Vector Arithmetic Exception (AEX). This bit is set by the vector processor when disabling itself due to an arithmetic exception. Information regarding the nature of the exception can be found in VAER. Writing a one to VPSR<AEX> clears VPSR<AEX> and VAER. Writing a zero to VPSR<AEX> has no effect.
<23:8>	R	0
<24>	R/W1C	Implementation-Specific Hardware Error (IMP). This bit is set by the vector processor when disabling itself due to an implementation-specific hardware error. Writing a one to VPSR<IMP> clears it. Writing a zero to VPSR<IMP> has no effect. \An implementation may choose not to implement VPSR<IMP>. In this case, writing VPSR<IMP> with either value must have no effect and must not generate any error. Also, its value when read must be zero.\
<25>	R/W1C	Illegal Vector Opcode (IVO). This bit is set by the vector processor when disabling itself due to receiving an illegal vector opcode. Writing a one to VPSR<IVO> clears it. Writing a zero to VPSR<IVO> has no effect. \An implementation may choose not to implement VPSR<IVO>. In this case, writing VPSR<IVO> with either value must have no effect and must not generate any error. Also, its value when read must be zero.\
<30:26>	R	0
<31>	R	Vector Processor Busy (BSY). When this bit is set, the vector processor is executing vector instructions. When it is clear, the vector processor is idle, or the vector processor has suspended instruction execution due to an asynchronous memory management exception or hardware error. Writing to VPSR<BSY> has no effect.

Table 13-2: Possible VPSR<3:0> Settings For MTPR

RLD	STS	RST	VEN	Meaning
0	0	0	0	Disable vector processor
0	0	0	1	Enable vector processor
0	0	1	0	Reset state and disable vector processor
0	0	1	1	Reset state and enable vector processor
0	1	0	0	Store state (must disable vector processor)
1	0	0	0	Reload state and disable vector processor
1	0	0	1	Reload state and then enable vector processor

Table 13-3: State of the Vector Processor

VEN	BSY	Meaning
0	0	The vector processor is not executing any instructions now, and either has no pending instructions or will not execute pending instructions. No more instructions should be sent.
0	1	The vector processor is executing at least one pending instruction. No more instructions should be sent.
1	0	The vector processor is not executing any instructions now, and either has no pending instructions or will not execute pending instructions. New instructions can be sent to the vector processor.
1	1	The vector processor is executing at least one instruction now. New instructions can be sent.

```
 3                                1 1                              0
 1                                6 5
+---------------------------------+---------------------------------+
|   vector dst register mask      |  exception condition summary    |
+---------------------------------+---------------------------------+
```

Figure 13-6 Vector Arithmetic Exception Register (VAER)

Table 13-4: VAER Exception Condition Summary Word Encoding

Bit	Exception Condition
<0>	Floating Underflow
<1>	Floating Divide by Zero
<2>	Floating Reserved Operand
<3>	Floating Overflow
<4>	0
<5>	Integer Overflow
<15:6>	0

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

Figure 13-7 Vector Memory Activity Check Register (VMAC)

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

Figure 13-8 Vector Translation Buffer Invalidate All Register (VTBIA)

```
 3
 1                                                         3 2   0
+---------------------------------------------------------+-----+
|                   virtual memory address                | SBZ |
+---------------------------------------------------------+-----+
```

Figure 13-9 Vector State Address Register (VSAR)

Table 13-5: IPR Assignments

Offset (Hex)	IPR
90	VPSR
91	VAER
92	VMAC
93	VTBIA
94	VSAR
95-9B	Reserved for vector architecture use
9C-9F	Reserved for vector implementation use

13.3 VECTOR INSTRUCTION FORMATS

Vector instructions use two-byte opcodes and normal VAX operand specifiers. For more information on VAX operand specifiers, refer to Chapter 2. The vector registers to be used by a vector instruction are specified by the vector control word operand. The MFVP, MTVP, and VSYNC instructions do not use a vector control word operand. The general format of the vector control word operand is shown in Figure 13-10. Table 13-6 describes the fields of the vector control word operand (cntrl). The actual format of the vector control word operand is instruction dependent. (Refer to the instruction descriptions later in this chapter for more detail.) The vector control word operand is passed by the VAX scalar processor to the vector processor.

The vector control word operand may determine some or all of the following:

o Enabling of masked operations.

o Enabling of floating underflow for floating-point instructions and integer overflow for integer operations.

o Which vector registers to use as sources/destinations.

o Which type of operation to perform (for the convert and compare instructions).

13.3.1 Masked Operations

Masked operations are enabled by the use of cntrl<15:14> of the vector control word operand. Cntrl<15> is the Masked Operations Enable bit (MOE), and cntrl<14> is the Match True/False bit (MTF). When cntrl<MOE> is set, masked operations are enabled. Only elements for which the corresponding Vector Mask Register (VMR) bit matches cntrl<MTF> are operated upon. If cntrl<MOE> is clear, all elements are operated upon. In either case, the Vector Length Register (VLR) limits the highest element operated upon.

Cntrl<MOE> should be zero for VMERGE and IOTA instructions; otherwise the results are UNPREDICTABLE. Both the Vector Mask Register (VMR) and the Match True/False bit (cntrl<MTF>) are always used by these instructions. VMERGE and IOTA operate upon vector register elements up to the value specified in VLR.

\ IMPLEMENTATION NOTE

It is anticipated that the FORTRAN vector compiler will generate masked operations frequently. Also, in a significant portion of these cases (25-50%), VMR will be set to all zeros or all ones. A performance improvement will result if the following steps are followed in the execution of a vector instruction that uses masked operations:

1. First determine if masked operations are enabled. If MOE EQL 1, go to step 2. If MOE EQL 0, go to step 3.

2. Check for the all-mismatch case. The all-mismatch case consists of two possibilities. The first occurs when MTF EQL 1 and VMR contains all zeros. The second occurs when MTF EQL 0 and VMR contains all ones. If the vector instruction reflects the all-mismatch case, then no elements will be processed by the vector instruction. If the vector instruction reflects the all-mismatch case, then exit, else go to step 3. There are restrictions on performing this all-mismatch case exit. Refer to section 14.3.5, Required Use of Memory Synchronization Instructions, for more information.

3. Execute the vector instruction.

This technique is complicated by the fact that VLR limits the number of elements operated upon. This can be managed in one of two ways:

1. Check only those VMR bits from 0 to VLR in step 2.

2. Check all the bits in VMR regardless of VLR. In this case, the performance improvement will be lost when VLR LSSU 64, and all VMR bits from 0 to VLR do not match MTF, and some number of VMR bits after VLR match MTF. This case may not degrade performance by that much because the strip-mining needed for loops longer than 64 elements guarantees that VLR EQLU 64 a high fraction of the time. \

13.3.2 Exception Enable Bit

The vector processor does not use the IV and FU bits in the PSL to enable integer overflow and floating underflow exception conditions. These exception conditions are enabled or disabled on a per instruction basis for vector integer and floating-point instructions by bit <13> in the vector control word operand (cntrl<EXC>). When cntrl<EXC> is set, floating underflow is enabled for vector floating point instructions, and integer overflow is enabled for vector integer instructions. When cntrl<EXC> is clear, floating underflow and integer overflow are disabled.

Note that for VLD/VGATH instructions bit<13> is used and labeled differently.

13.3.3 Modify Intent bit

By using the Modify Intent bit, software can indicate to the vector processor that a majority of the memory locations being loaded by a VLD/VGATH instruction will later be stored into by VST/VSCAT instructions -- and so become modified. When informed of software's intent to modify, some vector processor implementations can optimize the vector loads and stores performed on these locations.

The modify intent bit resides in bit<13> of the vector control word operand (cntrl<MI>) and is used only in VLD and VGATH instructions. A vector processor implementation is not required to implement cntrl<MI>.

For vector processors that implement cntrl<MI>, software uses the bit in a VLD or VGATH instruction in the following way:

- o By setting cntrl<MI> to 0, software indicates that less than a majority of the locations loaded by the VLD/VGATH will later be stored into by VST/VSCAT instructions.
- o By setting cntrl<MI> to 1, software indicates that a majority of the locations loaded by the VLD/VGATH will later be stored into by VST/VSCAT instructions.

Vector processors that do not implement cntrl<MI> ignore the setting of this bit in the control word for VLD and VGATH.

The results of VLD/VGATH and VST/VSCAT are unaffected by the setting of cntrl<MI>. This includes memory management, where access-checking is done with read intent for VLD/VGATH even if cntrl<MI> is set. However, incorrectly setting cntrl<MI> can prevent the optimization of these instructions.

\ IMPLEMENTATION NOTE

Here are some software guidelines for using cntrl<MI> on Rigel follow-on products (such as Mariah and NVAX):

1. At this moment, a majority of locations is 75% or more. This is subject to change and will be re-evaluated when there is sufficient hardware to test it on. So, presently, only set the bit when the entirety of the memory locations being read in overlap the entirety of the memory locations that will be written by 75% or more.

2. Initially, consider setting cntrl<MI> only for modifies whose overlap is known at compile time.

3. Stride does not matter in determining the overlap.

4. Only set cntrl<MI> when the modify (both load and store) is performed within the same vectorized loop.

5. In nested loops, only set cntrl<MI> for a modify (both load and store) which is performed completely within the innermost loop. \

13.3.4 Register Specifier Fields

The Va (cntrl<11:8>), Vb (cntrl<7:4>), and Vc (cntrl<3:0>) fields of the vector control word operand are generally used to select vector registers. Some vector instructions use these fields to encode other instruction-specific information as shown later in this section.

13.3.5 Vector Control Word Formats

Depending on the instruction, the vector control word can specify up to two vector registers as sources, and one vector register as a destination. Other information may be encoded in the vector control word, as shown in Figures 13-11a through 13-11l. Bits that are shown as "0" should be zero (SBZ). Execution of vector instructions with illegal, inconsistent, or unspecified control word fields produces UNPREDICTABLE results.

Figure 13-11a depicts the vector control word for VLDL and VLDQ.

Figure 13-11b depicts the vector control word for VSTL and VSTQ.

Figure 13-11c depicts the vector control word for VGATHL and VGATHQ.

Figure 13-11d depicts the vector control word for VSCATL and VSCATQ.

Figure 13-11e depicts the vector control word for VVADDL/F/D/G, VVSUBL/F/D/G, VVMULL/F/D/G, and VVDIVF/D/G.

Figure 13-11f depicts the vector control word for VVSLLL, VVSRLL, VVBISL, VVXORL, and VVBICL. Cntrl<EXC> should always be zero for these instructions, otherwise the results are UNPREDICTABLE.

Figure 13-11g depicts the vector control word for VVCMPL/F/D/G. The Vc field (cntrl<3:0>) is used to specify the compare function.

Figure 13-11h depicts the vector control word for VVCVT. The Va field (cntrl <11:8>) is used to specify the convert function.

Figure 13-11i depicts the vector control word for VVMERGE.

Figure 13-11j depicts the vector control word for VSADDL/F/D/G, VSSUBL/F/D/G, VSMULL/F/D/G, and VSDIVF/D/G.

Figure 13-11k depicts the vector control word for VSSLLL, VSSRLL, VSBISL, VSXORL, and VSBICL. Cntrl<EXC> should be zero for these instructions; otherwise, the results are UNPREDICTABLE.

Figure 13-11l depicts the vector control word for VSCMPL/F/D/G. The Vc field (cntrl<3:0>) is used to specify the compare function.

Figure 13-11m depicts the vector control word for VSMERGE.

Figure 13-11n depicts the vector control word for IOTA.

13.3.6 Restrictions on Operand Specifier Usage

Certain restrictions are placed on the addressing mode combinations usable within a single vector instruction. These combinations involve the logically inconsistent simultaneous use of a value as both a source operand (i.e., a .rw, .rl, or .rq operand) and an address. Specifically, if within the same instruction the contents of register Rn is used as both a part of a source operand and as an address in an addressing mode that modifies Rn (i.e., autodecrement, autoincrement, or autoincrement deferred), the value of the scalar source operand is UNPREDICTABLE.

Use of short literal mode for the scalar source operand of a vector floating-point instruction causes UNPREDICTABLE results.

If a Store Vector Register Data into Memory (VST) or Scatter Memory Data into Vector Register (VSCAT) instruction overwrites anything needed for calculation of the memory addresses to be written, the result of the VST or VSCAT is UNPREDICTABLE.

If the same vector register is used as both source and destination in a Gather Memory Data into Vector Register (VGATH) instruction, the result of the VGATH is UNPREDICTABLE.

When the addressing mode of the BASE operand used in a VLD, VST, VGATH, or VSCAT instruction is immediate, the results of the instruction are UNPREDICTABLE.

13.3.7 VAX Condition Codes

The vector instructions do not affect the condition codes in the PSL of the associated scalar processor.

13.3.8 Illegal Vector Opcodes

An illegal vector opcode is defined as a vector opcode to which no vector processor function is currently assigned. Opcodes that are not listed in table A-3 (in appendix A) as vector opcodes are neither decoded nor executed by the vector processor.

\It is likely that instructions can never be assigned to the illegal vector opcodes. Machines that implement block vector opcode decoding schemes report illegal vector opcodes by signaling a vector processor disabled fault with VPSR<IVO> set. Since the PC at which the vector processor disabled fault is reported is unpredictable, an emulator would be unable to determine which instruction to emulate.\

An implementation is permitted to report an illegal vector opcode in one of two ways:

1. Reserved-instruction fault. This is the recommended implementation.

2. Illegal vector opcode. The vector processor disables itself and sets VPSR<IVO>. The remainder of the vector processor state is left unmodified.

It is implementation-specific which way a particular illegal vector opcode is reported.

```
 1 1 1 1 1                                   1 1 1 1 1
 5 4 3 2 1    8 7    4 3    0                5 4 3 2 1    8 7    4 3    0
+-+-+-+-+-------+-------+-------+           +-+-+-+-+-------+-------+-------+
|M|M|E| |       |       |       |           |M|M|M| |       |       |       |
|O|T|X|0|  Va   |  Vb   |  Vc   |     or    |O|T|I|0|  Va   |  Vb   |  Vc   |
|E|F|C| |       |       |       |           |E|F| | |       |       |       |
+-+-+-+-+-------+-------+-------+           +-+-+-+-+-------+-------+-------+
```

Figure 13-10 Vector Control Word Operand (cntrl)

Table 13-6: Description of the Vector Control Word Operand

Extent	Description
<15>	MOE - Masked Operation Enable. This bit enables operations under the control of the Vector Mask Register (VMR) for vector instructions. When set, masked operations are enabled, and only elements for which the corresponding Vector Mask Register (VMR) bit matches cntrl<MTF> are operated on. If cntrl<MOE> is clear, all elements are operated upon. In either case, the Vector Length Register (VLR) limits the highest element operated upon.
<14>	MTF - Match True/False. When masked operations have been enabled (cntrl<MOE> EQL 1), only elements for which the corresponding Vector Mask Register (VMR) bit matches cntrl<MTF> are operated on. See above description. Cntrl<MTF> is also used by the VMERGE and IOTA instructions.
<13>	EXC - Exception Enable. Used only in vector integer and floating-point instructions to enable integer overflow and floating underflow, respectively.
<13>	MI - Modify Intent. Used only in VLD/VGATH instructions to indicate that a majority of the memory locations being loaded by the VLD or VGATH will later be stored into by VST/VSCAT instructions. This bit is optional to implement. See section 13.3.3, Modify Intent bit, for more details.
<12>	SBZ (Should Be Zero). \This bit is reserved for future use by the VSCAT instruction. Current implementations must ignore this bit and not produce UNPREDICTABLE results when it is set. Software should not set this bit.\
<11:8>	Va - This field selects the vector register to be used as the Va operand. For VVCVT, it specifies the convert function.
<7:4>	Vb - This field selects the vector register to be used as the Vb operand.
<3:0>	Vc - This field selects the vector register to be used as the Vc operand. For VCMP, it specifies the compare function.

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M|M| |       |       |dst/src|
|O|T|I|0|   0   |   0   |vec reg|
|E|F| | |       |       |  num  |
+-+-+-+-+-------+-------+-------+
```

a. Vector Control Word Format for VLDL and VLDQ.

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M| | |       |       |dst/src|
|O|T|0|0|   0   |   0   |vec reg|
|E|F| | |       |       |  num  |
+-+-+-+-+-------+-------+-------+
```

b. Vector Control Word Format for VSTL and VSTQ.

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M|M| |       |  src  |dst/src|
|O|T|I|0|   0   |vec reg|vec reg|
|E|F| | |       |  num  |  num  |
+-+-+-+-+-------+-------+-------+
```

c. Vector Control Word Format for VGATHL VGATHQ.

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M| | |       |  src  |dst/src|
|O|T|0|0|   0   |vec reg|vec reg|
|E|F| | |       |  num  |  num  |
+-+-+-+-+-------+-------+-------+
```

d. Vector Control Word Format for VSCATL and VSCATQ.

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M|E| | src1  | src2  |  dst  |
|O|T|X|0|vec reg|vec reg|vec reg|
|E|F|C| |  num  |  num  |  num  |
+-+-+-+-+-------+-------+-------+
```

e. Vector Control Word Format for VVADDL/F/D/G, VVSUBL/F/D/G, VVMULL/F/D/G, and VVDIVF/D/G.

Figure 13-11 Vector Control Word Format

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M| | | src1  | src2  |  dst  |
|O|T|0|0|vec reg|vec reg|vec reg|
|E|F| | |  num  |  num  |  num  |
+-+-+-+-+-------+-------+-------+
```

f. Vector Control Word Format for VVSLLL, VVSRLL, VVBISL, VVXORL, and VVBICL.

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M| | | src1  | src2  |  cmp  |
|O|T|0|0|vec reg|vec reg| func  |
|E|F| | |  num  |  num  |       |
+-+-+-+-+-------+-------+-------+
```

g. Vector Control Word Format for VVCMPL/F/D/G.

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M|E| |  cvt  |  src  |  dst  |
|O|T|X|0| func  |vec reg|vec reg|
|E|F|C| |       |  num  |  num  |
+-+-+-+-+-------+-------+-------+
```

h. Vector Control Word Format for VVCVT.

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
| |M| | | src1  | src2  |  dst  |
|0|T|0|0|vec reg|vec reg|vec reg|
| |F| | |  num  |  num  |  num  |
+-+-+-+-+-------+-------+-------+
```

i. Vector Control Word Format for VVMERGE.

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M|E| |       |  src  |  dst  |
|O|T|X|0|   0   |vec reg|vec reg|
|E|F|C| |       |  num  |  num  |
+-+-+-+-+-------+-------+-------+
```

j. Vector Control Word Format for VSADDL/F/D/G, VSSUBL/F/D/G, VSMULL/F/D/G, and VSDIVF/D/G.

Figure 13-11 Vector Control Word Format

```
 1 1 1 1 1
 5 4 3 2 1      8 7      4 3      0
+-+-+-+-+-+-------+-------+-------+
|M|M| | | |       |  src  |  dst  |
|O|T|0|0|    0    |vec reg|vec reg|
|E|F| | | |       |  num  |  num  |
+-+-+-+-+-+-------+-------+-------+
```

k. Vector Control Word Format for VSSLLL, VSSRLL, VSBISL, VSXORL, and VSBICL.

```
 1 1 1 1 1
 5 4 3 2 1      8 7      4 3      0
+-+-+-+-+-+-------+-------+-------+
|M|M| | | |       |  src  |  cmp  |
|O|T|0|0|    0    |vec reg| func  |
|E|F| | | |       |  num  |       |
+-+-+-+-+-+-------+-------+-------+
```

l. Vector Control Word Format for VSCMPL/F/D/G.

```
 1 1 1 1 1
 5 4 3 2 1      8 7      4 3      0
+-+-+-+-+-+-------+-------+-------+
| |M| | | |       |  src  |  dst  |
|0|T|0|0|    0    |vec reg|vec reg|
| |F| | | |       |  num  |  num  |
+-+-+-+-+-+-------+-------+-------+
```

m. Vector Control Word Format for VSMERGE.

```
 1 1 1 1 1
 5 4 3 2 1      8 7      4 3      0
+-+-+-+-+-+-------+-------+-------+
| |M| | | |       |       |  dst  |
|0|T|0|0|    0    |   0   |vec reg|
| |F| | | |       |       |  num  |
+-+-+-+-+-+-------+-------+-------+
```

n. Vector Control Word Format for IOTA.

Figure 13-11 Vector Control Word Format

13.4 VECTOR INSTRUCTION CLASSES

The vector instruction set is divided into 6 major sections:

1. Vector Memory
2. Vector Integer
3. Vector Logical and Shift
4. Vector Floating Point
5. Vector Edit
6. Miscellaneous operations

Within each major section, closely related instructions are grouped and described together using the same conventions as established in Chapter 3.

13.5 VECTOR MEMORY ACCESS INSTRUCTIONS

There are alignment, stride, address specifier context, and access mode considerations for the vector memory access instructions.

These instructions require their vector operands to be naturally aligned in memory. Longwords must be aligned on longword boundaries. Quadwords must be aligned on quadword boundaries. If any vector element is not naturally aligned in memory, an access control violation occurs. For further details, see section 14.2.1, Vector Memory Management Exception Handling. The scalar operands need not be naturally aligned in memory.

A vector's stride is defined as the number of memory locations (bytes) between the starting address of consecutive vector elements. A contiguous vector that has longword elements has a stride of four; a contiguous vector that has quadword elements has a stride of eight.

The base address specifier used by the vector memory access instructions is of byte context, regardless of the data type. Arrays are addressed as byte strings. Index values in array specifiers are multiplied by one, and the amount of autoincrement or autodecrement, when either of these modes is used, is one.

A vector memory access instruction is executed using the access mode in effect when the instruction is issued by the scalar processor.

13.5.1 Memory Instructions

VLD Load Memory Data into Vector Register

Format:

```
        opcode  cntrl.rw, base.ab, stride.rl
```

Vector Control Word Format:

```
         1 1 1 1 1
         5 4 3 2 1     8 7     4 3     0
        +-+-+-+-+-------+-------+-------+
        |M|M|M| |       |       |       |
        |O|T|I|0|   0   |   0   |  Vc   |
        |E|F| | |       |       |       |
        +-+-+-+-+-------+-------+-------+
```

Operation:

```
        addr <- base

        FOR i <- 0 TO VLR-1
          BEGIN
            IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
              BEGIN
                IF {addr unaligned} THEN
                   {Vector Alignment Exception}

                IF VLDL THEN
                   Vc[i] <- (addr)<31:0>
                IF VLDQ THEN
                   Vc[i] <- (addr)<63:0>
              END
            addr <- addr + stride              !Increment by stride
          END
```

Vector Processor Exceptions:

Access Control Violation
Translation Not Valid
Vector Alignment

Opcodes:

34FD	VLDL	Load Longword Vector from Memory to Vector Register
36FD	VLDQ	Load Quadword Vector from Memory to Vector Register

Description:

The source operand vector is fetched from memory and written to vector destination register Vc. The length of the vector is specified by VLR. The virtual address of the source vector is computed using the base address and the stride. The address of element i (0 LEQU i LEQU (VLR-1)) is computed as {base+{i*stride}}. The stride can be positive, negative, or zero.

In VLDL, bits <31:0> of each destination vector element receive the memory data and bits <63:32> are UNPREDICTABLE.

If any vector element operated upon is not naturally aligned in memory, a vector alignment exception occurs.

The results of VLD are unaffected by the setting of cntrl<MI>. For more details about the use of cntrl<MI>, see section 13.3.3, Modify Intent bit.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

An implementation may load the elements of the vector in any order, and more than once. When a vector processor memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

VGATH   Gather Memory Data into Vector Register

Format:

```
opcode cntrl.rw, base.ab
```

Vector Control Word Format:

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-+-------+-------+-------+
|M|M|M| |       |       |       |
|O|T|I|0|   0   |  Vb   |  Vc   |
|E|F| | |       |       |       |
+-+-+-+-+-+-------+-------+-------+
```

Operation:

```
FOR i <- 0 TO VLR-1
  BEGIN
    addr <- base + Vb[i]<31:0>
    IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
        IF {addr unaligned} THEN
           {Vector Alignment Exception}

        IF VGATHL THEN
           Vc[i] <- (addr)<31:0>
        IF VGATHQ THEN
           Vc[i] <- (addr)<63:0>
      END
  END
```

Vector Processor Exceptions:

Access Control Violation
Translation Not Valid
Vector Alignment

Opcodes:

35FD	VGATHL	Gather Longword Vector from Memory to Vector Register
37FD	VGATHQ	Gather Quadword Vector from Memory to Vector Register

Description:

The source operand vector is fetched from memory and written to vector destination register Vc. The length of the vector is specified by VLR. The virtual address of the vector is computed using the base address and the 32-bit offsets in vector register Vb. The address of element i (0 LEQU i LEQU (VLR-1)) is computed as {base+Vb[i]}. The 32-bit offset can be positive, negative, or zero.

In VGATHL, bits <31:0> of each destination vector element receive the memory data and bits <63:32> are UNPREDICTABLE.

If any vector element operated upon is not naturally aligned in memory, a vector alignment exception occurs.

The results of VGATH are unaffected by the setting of cntrl<MI>. For more details about the use of cntrl<MI>, see section 13.3.3, Modify Intent bit.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

An implementation may load the elements of the vector in any order, and more than once. When a vector processor memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

If the same vector register is used as both source and destination, the result of the VGATH is UNPREDICTABLE.

VST Store Vector Register Data into Memory

Format:

```
        opcode  cntrl.rw, base.ab, stride.rl
```

Vector Control Word Format:

```
         1 1 1 1 1
         5 4 3 2 1     8 7     4 3     0
        +-+-+-+-+-------+-------+-------+
        |M|M| | |       |       |       |
        |O|T|0|0|   0   |   0   |  Vc   |
        |E|F| | |       |       |       |
        +-+-+-+-+-------+-------+-------+
```

Operation:

```
        addr <- base

        FOR i <- 0 TO VLR-1
          BEGIN
            IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
              BEGIN
                IF {addr unaligned} THEN
                   {Vector Alignment Exception}

                IF VSTL THEN
                   (addr)<31:0> <- Vc[i]<31:0>
                IF VSTQ THEN
                   (addr)<63:0> <- Vc[i]
              END
            addr <- addr + stride                  !Increment by stride
          END
```

Vector Processor Exceptions:

Access Control Violation
Translation Not Valid
Vector Alignment
Modify

Opcodes:

9CFD	VSTL	Store Longword Vector from Vector Register to Memory
9EFD	VSTQ	Store Quadword Vector from Vector Register to Memory

Description:

The source operand in vector register Vc is written to memory. The length of the vector is specified by VLR. The virtual address of the destination vector is computed using the base address and the stride. The address of element i (0 LEQU i LEQU (VLR-1)) is computed as {base+{i*stride}}. The stride can be positive, negative, or zero.

If any vector element operated upon is not naturally aligned in memory, a vector alignment exception occurs.

For a non-zero stride value, an implementation may store the vector elements in parallel; therefore the order in which these elements are stored is unpredictable. Furthermore, if the non-zero stride causes result locations in memory to overlap, then the values stored in the overlapping result locations are also UNPREDICTABLE.

For a stride value of zero, the highest-numbered register element destined for the single memory location becomes the final value of that location. \ Note this does not require that all the elements destined for the location actually be written or written in order so long as the final result is the highest-numbered element.\

When a vector processor memory management exception occurs, it is UNPREDICTABLE whether the vector processor writes any result location for which an exception did not occur. If the fault condition can be eliminated by software and the instruction restarted then the vector processor will ensure that all destination locations are written.

If the destination vector overlaps the vector instruction control word, base, or stride operand, the result of the instruction is UNPREDICTABLE.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

VSCAT Scatter Vector Register Data into Memory

Format:

```
        opcode  cntrl.rw, base.ab
```

Vector Control Word Format:

```
         1 1 1 1 1
         5 4 3 2 1     8 7     4 3     0
        +-+-+-+-+-------+-------+-------+
        |M|M| | |       |       |       |
        |O|T|0|0|   0   |  Vb   |  Vc   |
        |E|F| | |       |       |       |
        +-+-+-+-+-------+-------+-------+
```

Operation:

```
        FOR i <- 0 TO VLR-1
          BEGIN
            addr <- base + Vb[i]<31:0>
            IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
              BEGIN
                IF {addr unaligned} THEN
                   {Vector Alignment Exception}

                IF VSCATL THEN
                   (addr)<31:0> <- Vc[i]<31:0>
                IF VSCATQ THEN
                   (addr)<63:0> <- Vc[i]
              END
          END
```

Vector Processor Exceptions:

Access Control Violation
Translation Not Valid
Vector Alignment
Modify

Opcodes:

9DFD	VSCATL	Scatter Longword Vector from Vector Register to Memory
9FFD	VSCATQ	Scatter Quadword Vector from Vector Register to Memory

Description:

The source vector operand Vc is written to memory. The length of the vector is specified by the VLR register. The virtual address of the destination vector is computed using the base address operand and the 32-bit offsets in vector register Vb. The address of element i (0 LEQU i LEQU (VLR-1)) is computed as {base+Vb[i]}. The 32-bit offset can be positive, negative, or zero.

If any vector element operated upon is not naturally aligned in memory, a vector alignment exception occurs.

An implementation may store the vector elements in parallel; therefore, the order in which elements are stored to different memory locations is UNPREDICTABLE. In the case where multiple elements are destined for the same memory location, the highest-numbered element among them becomes the final value of that location. \ Note this does not require that all the elements destined for the same location actually be written or written in order so long as the final result is the highest-numbered element. This requirement was added so that the Fortran Compiler could better vectorize indirect addressing.\

\ Cntrl<12> is reserved for future use by the VSCAT instruction. Implementations must ignore cntrl<12> and software should not set it. In some future implementation, this bit may be used to allow VSCAT to store all elements out of order.\

When a vector processor memory management exception occurs, it is UNPREDICTABLE whether the vector processor writes any result location for which an exception did not occur. If the fault condition can be eliminated by software and the instruction restarted then the vector processor will ensure that all destination locations are written.

If the destination vector overlaps the vector instruction control word or base operand, the result of the instruction is UNPREDICTABLE.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

13.6 VECTOR INTEGER INSTRUCTIONS

VADDL Vector Integer Add

Format:

```
        opcode  cntrl.rw                  ! Vector + Vector
        opcode  cntrl.rw, addend.rl       ! Scalar + Vector
```

Vector Control Word Format:

```
 1 1 1 1 1
 5 4 3 2 1      8 7      4 3      0
+-+-+-+-+--------+--------+--------+
|M|M|E| |  Va    |        |        |
|O|T|X|0|  or    |   Vb   |   Vc   |
|E|F|C| |  0     |        |        |
+-+-+-+-+--------+--------+--------+
```

Operation:

```
FOR i <- 0 TO VLR-1
  IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
    BEGIN
      IF VVADDL THEN
         Vc[i]<31:0> <- Va[i]<31:0> + Vb[i]<31:0>
      IF VSADDL THEN
         Vc[i]<31:0> <- addend + Vb[i]<31:0>
    END
```

Vector Processor Exceptions:

Integer Overflow

Opcodes:

80FD	VVADDL	Vector Vector Add Longword
81FD	VSADDL	Vector Scalar Add Longword

Description:

The scalar addend or Va operand is added, element-wise, to vector register Vb and the 32-bit sum is written to vector register Vc. Only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by VLR.

If integer overflow is detected and cntrl<EXC> is set, the exception type and destination register number are recorded in the Vector Arithmetic Exception Register and the vector operation is allowed to complete. On integer overflow, the low-order 32 bits of the true result are stored in the destination element.

VCMPL Vector Integer Compare

Format:

```
opcode   cntrl.rw                   ! Vector - Vector
opcode   cntrl.rw, src.rl           ! Scalar - Vector
```

Vector Control Word Format:

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-+-------+-------+-------+
|M|M| | |  Va    |       |  cmp  |
|O|T|0|0|  or    |  Vb   |  func |
|E|F| | |  0     |       |       |
+-+-+-+-+-+-------+-------+-------+
```

The condition being tested is determined by cntrl<2:0> as follows:

0	Greater Than
1	Equal
2	Less Than
3	Reserved
4	Less Than or Equal
5	Not Equal
6	Greater Than or Equal
7	Reserved

Vector integer compare instructions that specify reserved values of cntrl<2:0> produce UNPREDICTABLE results.

Cntrl<3> should be zero; if it is set, the results of the instruction are UNPREDICTABLE.

Operation:

```
FOR i <- 0 TO VLR-1
  IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
    BEGIN
      IF VVCMPL THEN
        IF Va[i]<31:0> SIGNED_RELATION Vb[i]<31:0> THEN
           VMR<i> <- 1
        ELSE
           VMR<i> <- 0

      IF VSCMPL THEN
        IF src SIGNED_RELATION Vb[i]<31:0> THEN
           VMR<i> <- 1
        ELSE
           VMR<i> <- 0
    END
```

Vector Processor Exceptions:

None

Opcodes:

C0FD	VVCMPL	Vector Vector Compare Longword
C1FD	VSCMPL	Vector Scalar Compare Longword

Description:

The scalar or Va operand is compared, element-wise, with vector register Vb. The length of the vector is specified by VLR. For each element comparison, if the specified relationship is true, the Vector Mask Register bit (VMR<i>) corresponding to the vector element is set to one; otherwise, it is cleared. If cntrl<MOE> is set, VMR bits corresponding to elements that do not match cntrl<MTF> are left unchanged. VMR bits beyond the vector length are left unchanged. Only bits <31:0> of each vector element participate in the operation.

VMULL Vector Integer Multiply

Format:

```
        opcode  cntrl.rw                    ! Vector * Vector
        opcode  cntrl.rw, mulr.rl           ! Scalar * Vector
```

Vector Control Word Format:

```
         1 1 1 1 1
         5 4 3 2 1     8 7     4 3     0
        +-+-+-+-+-------+-------+-------+
        |M|M|E| |  Va   |       |       |
        |O|T|X|0|  or   |  Vb   |  Vc   |
        |E|F|C| |  0    |       |       |
        +-+-+-+-+-------+-------+-------+
```

Operation:

```
        FOR i <- 0 TO VLR-1
          IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
            BEGIN
              IF VVMULL THEN
                 Vc[i]<31:0> <- Va[i]<31:0> * Vb[i]<31:0>
              IF VSMULL THEN
                 Vc[i]<31:0> <- mulr * Vb[i]<31:0>
            END
```

Vector Processor Exceptions:

Integer Overflow

Opcodes:

A0FD	VVMULL	Vector Vector Multiply Longword
A1FD	VSMULL	Vector Scalar Multiply Longword

Description:

The scalar multiplier or vector operand Va is multiplied, element-wise, by vector operand Vb and the least significant 32 bits of the signed 64-bit product are written to vector register Vc. Only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by VLR.

If integer overflow is detected and cntrl<EXC> is set, the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register and the vector operation is allowed to complete. On integer overflow, the low-order 32 bits of the true result are stored in the destination element.

VSUBL Vector Integer Subtract

Format:

```
        opcode  cntrl.rw                    ! Vector - Vector
        opcode  cntrl.rw, min.rl            ! Scalar - Vector
```

Vector Control Word Format:

```
        1 1 1 1 1
        5 4 3 2 1     8 7     4 3     0
        +-+-+-+-+-------+-------+-------+
        |M|M|E| |  Va   |       |       |
        |O|T|X|0|  or   |  Vb   |  Vc   |
        |E|F|C| |  0    |       |       |
        +-+-+-+-+-------+-------+-------+
```

Operation:

```
        FOR i <- 0 TO VLR-1
          IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
            BEGIN
              IF VVSUBL THEN
                 Vc[i]<31:0> <- Va[i]<31:0> - Vb[i]<31:0>
              IF VSSUBL THEN
                 Vc[i]<31:0> <- min - Vb[i]<31:0>
            END
```

Vector Processor Exceptions:

Integer Overflow

Opcodes:

88FD	VVSUBL	Vector Vector Subtract Longword
89FD	VSSUBL	Vector Scalar Subtract Longword

Description:

The vector operand Vb is subtracted, element-wise, from the scalar minuend or vector operand Va. The 32-bit difference is written to vector register Vc. Only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by VLR.

If integer overflow is detected and cntrl<EXC> is set, the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register and the vector operation is allowed to complete. On integer overflow, the low-order 32 bits of the true result are stored in the destination element.

13.7 VECTOR LOGICAL AND SHIFT INSTRUCTIONS

Vector Logical Functions

Format:

```
        opcode  cntrl.rw                  ! Vector op Vector
        opcode  cntrl.rw, src.rl          ! Vector op Scalar
```

Vector Control Word Format:

```
         1 1 1 1 1
         5 4 3 2 1      8 7     4 3     0
        +-+-+-+-+-------+-------+-------+
        |M|M| | |  Va   |       |       |
        |O|T|0|0|  or   |  Vb   |  Vc   |
        |E|F| | |  0    |       |       |
        +-+-+-+-+-------+-------+-------+
```

Operation:

```
        FOR i <- 0 TO VLR-1
          IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
            BEGIN
              IF VVBISL THEN
                 Vc[i]<31:0> <- Va[i]<31:0> OR Vb[i]<31:0>
              IF VSBISL THEN
                 Vc[i]<31:0> <- src OR Vb[i]<31:0>
              IF VVXORL THEN
                 Vc[i]<31:0> <- Va[i]<31:0> XOR Vb[i]<31:0>
              IF VSXORL THEN
                 Vc[i]<31:0> <- src XOR Vb[i]<31:0>
              IF VVBICL THEN
                 Vc[i]<31:0> <- {NOT Va[i]<31:0>} AND Vb[i]<31:0>
              IF VSBICL THEN
                 Vc[i]<31:0> <- {NOT src} AND Vb[i]<31:0>
              Vc[i]<63:32> <- Vb[i]<63:32>
            END
```

Vector Processor Exceptions:

None

Opcodes:

C8FD	VVBISL	Vector Vector Bit Set Longword
E8FD	VVXORL	•Vector Vector Exclusive-OR Longword
CCFD	VVBICL	Vector Vector Bit Clear Longword
C9FD	VSBISL	Vector Scalar Bit Set Longword
E9FD	VSXORL	Vector Scalar Exclusive-OR Longword
CDFD	VSBICL	Vector Scalar Bit Clear Longword

Description:

The scalar src or vector operand Va is combined, element-wise, using the specified boolean function, with vector register Vb and the result is written to vector register Vc. Only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the elements of Vb are written into bits <63:32> of the corresponding elements of Vc. The length of the vector is specified by VLR.

\The logical operations copy bits <63:32> of Vb into Vc to allow math library routines to modify the sign and exponent fields of D_floating and G_floating data.\

VSL Vector Shift Logical

Format:

```
opcode  cntrl.rw                ! Vector Shift Count
opcode  cntrl.rw, cnt.rl        ! Scalar Shift Count
```

Vector Control Word Format:

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-+-------+-------+-------+
|M|M| | |  Va     |       |       |
|O|T|0|0|  or     |  Vb   |  Vc   |
|E|F| | |  0      |       |       |
+-+-+-+-+-+-------+-------+-------+
```

Operation:

```
FOR i <- 0 TO VLR-1
  IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
    BEGIN
      IF VVSRLL THEN
        Vc[i]<31:0> <- RIGHT_SHIFT(Vb[i]<31:0>, Va[i]<4:0>)
      IF VVSLLL THEN
        Vc[i]<31:0> <- LEFT_SHIFT(Vb[i]<31:0>,  Va[i]<4:0>)

      IF VSSRLL THEN
        Vc[i]<31:0> <- RIGHT_SHIFT(Vb[i]<31:0>, cnt<4:0>)
      IF VSSLLL THEN
        Vc[i]<31:0> <- LEFT_SHIFT(Vb[i]<31:0>,  cnt<4:0>)
    END
```

Vector Processor Exceptions:

None

Opcodes:

E0FD	VVSRLL	Vector Vector Shift Right Logical Longword
E4FD	VVSLLL	Vector Vector Shift Left Logical Longword
E1FD	VSSRLL	Vector Scalar Shift Right Logical Longword
E5FD	VSSLLL	Vector Scalar Shift Left Logical Longword

Description:

Each element in vector register Vb is shifted logically left or right 0 to 31 bits as specified by a scalar count operand or vector register Va. The shifted results are written to vector register Vc. Zero bits are propagated into the vacated bit positions. Only bits <4:0> of the count operand and bits <31:0> of each Vb element participate in the operation. Bits <63:32> of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by VLR.

13.8 VECTOR FLOATING-POINT INSTRUCTIONS

The VAX vector architecture provides instructions for operating on F_floating, D_floating, and G_floating operand formats. The floating-point arithmetic instructions are add, subtract, compare, multiply, and divide. Data conversion instructions are provided to convert operands between D_floating, G_floating, F_floating, and longword integer.

Rounding is performed using standard VAX rounding rules. The accuracy of the vector floating-point instructions matches that of the scalar floating-point instructions. Refer to the section on Floating Point Instructions in Chapter 3 for more information.

13.8.1 Vector Floating-Point Exception Conditions

All vector floating-point exception conditions occur asynchronously with respect to the scalar processor. These exception conditions do not interrupt the scalar processor. If the exception condition is enabled, then the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register, and a reserved operand in the format of the instruction's data type is written into the destination register element. Encoded in this reserved operand is the exception condition type. After recording the exception and writing the appropriate result into the destination register element, the instruction encountering the exception continues executing to completion.

If a vector convert floating to integer instruction encounters a source element that is a reserved operand, an UNPREDICTABLE result rather than a reserved operand is written into the destination register element.

Figure 13-12 shows the encoding of the reserved operand that is written for vector floating-point exceptions. Consistent with the definition of a reserved operand (as defined in section 3.9, Floating-Point Instructions) the sign bit (bit <15>) is one and the exponent (bits <14:7> for F_floating and D_floating, and bits <14:4> for G_floating) is zero. When the reserved operand is written in F_floating or D_floating format, bits <6:4> are also zero. \By making bits <6:4> to be zero for these reserved operands, Architectural Verification testing is simplified because all reserved operands that are written for vector floating-point exceptions have bits <14:4> as zero.\ The exception condition type (ETYPE) is encoded in bits <3:0>, as shown in Table 13-7. If a reserved operand is divided by zero, both ETYPE bits may be set. The state of all other bits in the result (denoted with an "x") is UNPREDICTABLE.

If the floating underflow exception condition is suppressed by cntrl<EXC>, a zero result is written to the destination register element and no further action is taken. Floating overflow, floating divide by zero, and floating reserved operand are always enabled.

```
    3                   1 1 1
    1                   6 5 4             7 6   4 3     0
   +---------------------+-+---------------+-----+-------+
   |xxxxxxxxxxxxxxxxxxxxx|1|       0       |  0  | ETYPE | :Vc[i]<31:0>
   +---------------------+-+---------------+-----+-------+
```

a. F_floating

```
    3                   1 1 1
    1                   6 5 4             7 6   4 3     0
   +---------------------+-+---------------+-----+-------+
   |xxxxxxxxxxxxxxxxxxxxx|1|       0       |  0  | ETYPE | :Vc[i]<31:0>
   +---------------------+-+---------------+-----+-------+
   |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| :Vc[i]<63:32>
   +-----------------------------------------------------+
```

b. D_floating

```
    3                   1 1 1
    1                   6 5 4                   4 3     0
   +---------------------+-+---------------------+-------+
   |xxxxxxxxxxxxxxxxxxxxx|1|          0          | ETYPE | :Vc[i]<31:0>
   +---------------------+-+---------------------+-------+
   |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| :Vc[i]<63:32>
   +-----------------------------------------------------+
```

c. G_floating

Figure 13-12 Encoding of the Reserved Operand

Table 13-7: Encoding of the Exception Condition Type (ETYPE)

Bit	Exception Condition Type
<0>	Floating Underflow
<1>	Floating Divide by Zero
<2>	Floating Reserved Operand
<3>	Floating Overflow

13.8.2 Floating-point Instructions

VADD Vector Floating Add

Format:

```
opcode  cntrl.rw                    ! Vector + Vector
opcode  cntrl.rw, addend.rl         ! Scalar + Vector (F)
opcode  cntrl.rw, addend.rq         ! Scalar + Vector (D and G)
```

Vector Control Word Format:

```
 1 1 1 1 1
 5 4 3 2 1      8 7      4 3      0
+-+-+-+-+--------+--------+--------+
|M|M|E| |   Va   |        |        |
|O|T|X|0|   or   |   Vb   |   Vc   |
|E|F|C| |   0    |        |        |
+-+-+-+-+--------+--------+--------+
```

Operation:

```
FOR i <- 0 TO VLR-1
  IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
    BEGIN
      IF VVADDF THEN
         Vc[i]<31:0> <- Va[i]<31:0> + Vb[i]<31:0>
      IF VSADDF THEN
         Vc[i]<31:0> <- addend + Vb[i]<31:0>

      IF VVADDD OR VVADDG THEN
         Vc[i] <- Va[i] + Vb[i]
      IF VSADDD OR VSADDG THEN
         Vc[i] <- addend + Vb[i]
    END
```

Vector Processor Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

84FD	VVADDF	Vector Vector Add F_Floating
85FD	VSADDF	Vector Scalar Add F_Floating
86FD	VVADDD	Vector Vector Add D_Floating
87FD	VSADDD	Vector Scalar Add D_Floating
82FD	VVADDG	Vector Vector Add G_Floating
83FD	VSADDG	Vector Scalar Add G_Floating

Description:

The source addend or vector operand Va is added, element-wise, to vector register Vb and the sum is written to vector register Vc. The length of the vector is specified by VLR.

In VxADDF, only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl<EXC> is set or if a floating overflow or floating reserved operand occurs, an encoded reserved operand is stored as the result and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register. The vector operation is then allowed to complete. If cntrl<EXC> is clear, zero is written to the destination element when an exponent underflow occurs and no other action is taken.

VCMP   Vector Floating Compare

Format:

```
        opcode  cntrl.rw                  ! Vector - Vector
        opcode  cntrl.rw, src.rl          ! Scalar - Vector (F)
        opcode  cntrl.rw, src.rq          ! Scalar - Vector (D and G)
```

Vector Control Word Format:

```
         1 1 1 1 1
         5 4 3 2 1       8 7     4 3      0
        +-+-+-+-+-+-------+-------+-------+
        |M|M| | |  Va     |       |  cmp  |
        |O|T|0|0|  or     |  Vb   |  func |
        |E|F| | |  0      |       |       |
        +-+-+-+-+-+-------+-------+-------+
```

The condition being tested is determined by cntrl<2:0> as follows:

0	Greater Than
1	Equal
2	Less Than
3	Reserved
4	Less Than or Equal
5	Not Equal
6	Greater Than or Equal
7	Reserved

Vector floating compare instructions that specify reserved values of cntrl<2:0> produce UNPREDICTABLE results.

Cntrl<3> should be zero; if it is set, the results of the instruction are UNPREDICTABLE.

Operation:

```
FOR i <- 0 TO VLR-1
  IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
    BEGIN
      IF VVCMPF THEN
        IF Va[i]<31:0> SIGNED_RELATION Vb[i]<31:0> THEN
           VMR<i> <- 1
        ELSE
           VMR<i> <- 0
      IF VVCMPD OR VVCMPG THEN
        IF Va[i] SIGNED_RELATION Vb[i] THEN
           VMR<i> <- 1
        ELSE
           VMR<i> <- 0
      IF VSCMPF THEN
        IF src SIGNED_RELATION Vb[i]<31:0> THEN
           VMR<i> <- 1
        ELSE
```

```
            VMR<i> <- 0
         IF VSCMPD OR VSCMPG THEN
           IF src SIGNED_RELATION Vb[i] THEN
              VMR<i> <- 1
           ELSE
              VMR<i> <- 0
       END
```

Vector Processor Exceptions:

Floating Reserved Operand

Opcodes:

C4FD	VVCMPF	Vector Vector Compare F_floating
C5FD	VSCMPF	Vector Scalar Compare F_floating
C6FD	VVCMPD	Vector Vector Compare D_floating
C7FD	VSCMPD	Vector Scalar Compare D_floating
C2FD	VVCMPG	Vector Vector Compare G_floating
C3FD	VSCMPG	Vector Scalar Compare G_floating

Description:

The scalar or vector operand Va is compared, element-wise, with vector register Vb. The length of the vector is specified by VLR. For each element comparison, if the specified relationship is true, the Vector Mask Register bit (VMR<i>) corresponding to the vector element is set to one, otherwise it is cleared. If cntrl<MOE> is set, VMR bits corresponding to elements that do not match cntrl<MTF> are left unchanged. VMR bits beyond the vector length are left unchanged. If an element being compared is a reserved operand, VMR<i> is UNPREDICTABLE. In VxCMPF, only bits <31:0> of each vector element participate in the operation.

If a floating reserved operand exception occurs, the exception condition type is recorded in the VAER and the vector operation is allowed to complete.

Note that for this instruction, no bits are set in the VAER destination register mask when an exception occurs.

VVCVT   Vector Convert

Format:

    opcode cntrl.rw

Vector Control Word Format:

```
 1 1 1 1 1
 5 4 3 2 1      8 7      4 3      0
+-+-+-+-+-+-------+-------+-------+
|M|M|E| | cvt     |       |       |
|O|T|X|0| func    |  Vb    |  Vc    |
|E|F|C| |         |       |       |
+-+-+-+-+-+-------+-------+-------+
```

The convert operation is determined by cntrl<11:8> as shown below:

cntrl<11:8>	Interpretation
1 1 1 1	CVTRGL - Convert Rounded G_Floating to Longword
1 1 1 0	Reserved
1 1 0 1	CVTGF - Convert Rounded G_Floating to F_Floating
1 1 0 0	CVTGL - Convert Truncated G_Floating to Longword
1 0 1 1	Reserved
1 0 1 0	CVTRDL - Convert Rounded D_Floating to Longword
1 0 0 1	CVTDF - Convert Rounded D_Floating to F_Floating
1 0 0 0	CVTDL - Convert Truncated D_Floating to Longword
0 1 1 1	CVTFG - Convert F_Floating to G_Floating (exact)
0 1 1 0	CVTFD - Convert F_Floating to D_Floating (exact)
0 1 0 1	CVTRFL - Convert Rounded F_Floating to Longword
0 1 0 0	CVTFL - Convert Truncated F_Floating to Longword
0 0 1 1	CVTLG - Convert Longword to G_Floating (exact)
0 0 1 0	CVTLD - Convert Longword to D_Floating (exact)
0 0 0 1	CVTLF - Convert Rounded Longword to F_Floating
0 0 0 0	Reserved

Vector convert instructions that specify reserved values of cntrl<11:8> produce UNPREDICTABLE results.

Operation:

```
FOR i <- 0 TO VLR-1
  IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
     Vc[i] <- {conversion of Vb[i]}
```

Vector Processor Exceptions:

    Floating Overflow
    Floating Reserved Operand
    Floating Underflow
    Integer Overflow

Opcode:

ECFD VVCVT Vector Convert

Description:

The vector elements in vector register Vb are converted and results are written to vector register Vc. Cntrl<11:8> specifies the conversion to be performed. The length of the vector is specified by VLR. Bits <63:32> of Vc are UNPREDICTABLE for instructions that convert from D_floating or G_floating to F_floating or longword. When CVTRGL, CVTRDL, and CVTRFL round, the rounding is done in sign magnitude, before conversion to two's complement.

If an integer overflow occurs when cntrl<EXC> is set, the low-order 32 bits of the true result are written to the destination element as the result, and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register. The vector operation is then allowed to complete. If integer overflow occurs when cntrl<EXC> is clear, the low-order 32 bits of the true result are written to the destination element, and no other action is taken.

For vector convert floating to integer, where the source element is a reserved operand, the value written to the destination element is UNPREDICTABLE. In addition, the exception type and destination register number are recorded in the Vector Arithmetic Exception Register. The vector operation is then allowed to complete.

For vector convert floating to floating instructions, if floating underflow occurs when cntrl<EXC> is clear, zero is written to the destination element, and no other action is taken. The vector operation is then allowed to complete.

For vector convert floating to floating instructions, if floating underflow occurs with cntrl<EXC> set or if a floating overflow or reserved operand occurs, an encoded reserved operand is written to the destination element, and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register. The vector operation is then allowed to complete.

VDIV Vector Floating Divide

Format:

```
        opcode  cntrl.rw                    ! Vector / Vector
        opcode  cntrl.rw, divd.rl           ! Scalar / Vector (F)
        opcode  cntrl.rw, divd.rq           ! Scalar / Vector (D and G)
```

Vector Control Word Format:

```
         1 1 1 1 1
         5 4 3 2 1      8 7      4 3      0
        +-+-+-+-+-------+--------+--------+
        |M|M|E| |  Va   |        |        |
        |O|T|X|0|  or   |   Vb   |   Vc   |
        |E|F|C| |  0    |        |        |
        +-+-+-+-+-------+--------+--------+
```

Operation:

```
FOR i <- 0 TO VLR-1
  IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
    BEGIN
      IF VVDIVF THEN
         Vc[i]<31:0> <- Va[i]<31:0> / Vb[i]<31:0>
      IF VSDIVF THEN
         Vc[i]<31:0> <- divd / Vb[i]<31:0>

      IF VVDIVD OR VVDIVG THEN
         Vc[i] <- Va[i] / Vb[i]
      IF VSDIVD OR VSDIVG THEN
         Vc[i] <- divd / Vb[i]
    END
```

Vector Processor Exceptions:

Floating Divide by Zero
Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

ACFD	VVDIVF	Vector Vector Divide F_floating
ADFD	VSDIVF	Vector Scalar Divide F_floating
AEFD	VVDIVD	Vector Vector Divide D_floating
AFFD	VSDIVD	Vector Scalar Divide D_floating
AAFD	VVDIVG	Vector Vector Divide G_floating
ABFD	VSDIVG	Vector Scalar Divide G_floating

Description:

The scalar dividend or vector register Va is divided, element-wise, by the divisor in vector register Vb and the quotient is written to vector register Vc. The length of the vector is specified by VLR.

In VxDIVF, only bits <31:0> of each vector element participate in the operation; bits <63:32> of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl<EXC> is set or if a floating overflow, divide by zero or reserved operand occurs, an encoded reserved operand is stored as the result and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register. The vector operation is then allowed to complete. If cntrl<EXC> is clear, zero is written to the destination element when an exponent underflow occurs and no other action is taken.

VMUL   Vector Floating Multiply

Format:

```
opcode  cntrl.rw                   ! Vector * Vector
opcode  cntrl.rw, mulr.rl          ! Scalar * Vector (F)
opcode  cntrl.rw, mulr.rq          ! Scalar * Vector (D and G)
```

Vector Control Word Format:

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
|M|M|E| |  Va   |       |       |
|O|T|X|0|  or   |  Vb   |  Vc   |
|E|F|C| |  0    |       |       |
+-+-+-+-+-------+-------+-------+
```

Operation:

```
FOR i <- 0 TO VLR-1
  IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
    BEGIN
      IF VVMULF THEN
         Vc[i]<31:0> <- Va[i]<31:0> * Vb[i]<31:0>
      IF VSMULF THEN
         Vc[i]<31:0> <- mulr * Vb[i]<31:0>

      IF VVMULD OR VVMULG THEN
         Vc[i] <- Va[i] * Vb[i]
      IF VSMULD OR VSMULG THEN
         Vc[i] <- mulr * Vb[i]
    END
```

Vector Processor Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

A4FD	VVMULF	Vector Vector Multiply F_floating
A5FD	VSMULF	Vector Scalar Multiply F_floating
A6FD	VVMULD	Vector Vector Multiply F_floating
A7FD	VSMULD	Vector Scalar Multiply D_floating
A2FD	VVMULG	Vector Vector Multiply G_floating
A3FD	VSMULG	Vector Scalar Multiply G_floating

Description:

The multiplicand in vector register Vb is multiplied, element-wise, by the scalar multiplier or vector operand Va and the product is written to vector register Vc. The length of the vector is specified by VLR.

In VxMULF, only bits <31:0> of each vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl<EXC> is set or if a floating overflow or reserved operand occurs, an encoded reserved operand is stored as the result and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register. The vector operation is then allowed to complete. If cntrl<EXC> is clear, zero is written to the destination element when an exponent underflow occurs and no other action is taken.

VSUB   Vector Floating Subtract

Format:

```
        opcode  cntrl.rw                    ! Vector - Vector
        opcode  cntrl.rw, min.rl            ! Scalar - Vector (F)
        opcode  cntrl.rw, min.rq            ! Scalar - Vector (D and G)
```

Vector Control Word Format:

```
         1 1 1 1 1
         5 4 3 2 1     8 7     4 3     0
        +-+-+-+-+-------+-------+-------+
        |M|M|E| |  Va   |       |       |
        |O|T|X|0|  or   |  Vb   |  Vc   |
        |E|F|C| |  0    |       |       |
        +-+-+-+-+-------+-------+-------+
```

Operation:

```
        FOR i <- 0 TO VLR-1
          IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
            BEGIN
              IF VVSUBF THEN
                 Vc[i]<31:0> <- Va[i]<31:0> - Vb[i]<31:0>
              IF VSSUBF THEN
                 Vc[i]<31:0> <- min - Vb[i]<31:0>

              IF VVSUBD OR VVSUBG THEN
                 Vc[i] <- Va[i] - Vb[i]
              IF VSSUBD OR VSSUBG THEN
                 Vc[i] <- min - Vb[i]
            END
```

Vector Processor Exceptions:

Floating Overflow
Floating Reserved Operand
Floating Underflow

Opcodes:

8CFD	VVSUBF	Vector Vector Subtract F_floating
8DFD	VSSUBF	Vector Scalar Subtract F_floating
8EFD	VVSUBD	Vector Vector Subtract D_floating
8FFD	VSSUBD	Vector Scalar Subtract D_floating
8AFD	VVSUBG	Vector Vector Subtract G_floating
8BFD	VSSUBG	Vector Scalar Subtract G_floating

Description:

Vector register Vb is subtracted, element-wise, from the scalar minuend or vector register Va and the difference is written to vector register Vc. The length of the vector is specified by VLR.

In VxSUBF, only bits <31:0> of each vector element participate in the operation; bits <63:32> of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl<EXC> is set or if a floating overflow or reserved operand occurs, an encoded reserved operand is stored as the result and the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register. The vector operation is then allowed to complete. If cntrl<EXC> is clear, zero is written to the destination element when an exponent underflow occurs and no other action is taken.

13.9 VECTOR EDIT INSTRUCTIONS

VMERGE Vector Merge

Format:

```
        opcode  cntrl.rw                 ! Vector Vector Merge
        opcode  cntrl.rw, src.rq         ! Vector Scalar Merge
```

Vector Control Word Format:

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-------+-------+-------+
| |M| | |  Va   |       |       |
|0|T|0|0|  or   |  Vb   |  Vc   |
| |F| | |  0    |       |       |
+-+-+-+-+-------+-------+-------+
```

Operation:

```
FOR i <- 0 TO VLR-1
  BEGIN
    IF VVMERGE THEN
      IF {VMR<i> EQL MTF} THEN
         Vc[i] <- Va[i]
      ELSE
         Vc[i] <- Vb[i]

    IF VSMERGE THEN
      IF {VMR<i> EQL MTF} THEN
         Vc[i] <- src
      ELSE
         Vc[i] <- Vb[i]
  END
```

Vector Processor Exceptions:

None

Opcodes:

EEFD	VVMERGE	Vector Vector Merge
EFFD	VSMERGE	Vector Scalar Merge

Description:

The scalar src or vector operand Va is merged, element-wise, with vector register Vb and the resulting vector is written to vector register Vc. The length of the vector operation is specified by VLR.

For each vector element, i, if the corresponding Vector Mask Register bit (VMR<i>) matches cntrl<MTF>, src or Va[i] is written to the destination vector element Vc[i]. If VMR<i> does not match cntrl<MTF>, Vb[i] is written to the destination vector element.

IOTA Generate Compressed Iota Vector

Format:

```
opcode  cntrl.rw, stride.rl
```

Vector Control Word Format:

```
 1 1 1 1 1
 5 4 3 2 1     8 7     4 3     0
+-+-+-+-+-+-------+-------+-------+
| |M| | | |       |       |       |
|0|T|0|0|    0    |   0   |  Vc   |
| |F| | | |       |       |       |
+-+-+-+-+-+-------+-------+-------+
```

Operation:

```
j   <- 0
tmp <- 0
FOR i <- 0 TO VLR-1
  BEGIN
    IF {VMR<i> EQL MTF} THEN
      BEGIN
        Vc[j]<31:0> <- tmp<31:0>
        j <- j + 1
      END
    tmp <- tmp + stride
  END
VCR <- j                              !return vector count
```

Vector Processor Exceptions:

None

Opcode:

EDFD IOTA Generate Compressed Iota Vector

Description:

IOTA constructs a vector of offsets for use by the vector gather/scatter instructions VGATH and VSCAT.

IOTA first generates an iota vector of length VLR using the stride operand. An iota vector is a vector whose first element is zero and whose subsequent elements are spaced by the stride increment. The stride can be positive, negative, or zero. For example,

```
0*stride, 1*stride, 2*stride, 3*stride, ..., {VLR-1}*stride
```

The iota vector is then compressed using the contents of the Vector Mask Register. Elements of the iota vector for which the corresponding Vector Mask Register bit matches cntrl<MTF> are written in contiguous elements of the destination vector register Vc. Only bits <31:0> of each iota and destination vector element participate in the operation. Bits <63:32> of the destination vector elements are UNPREDICTABLE.

The number of elements written to Vc is returned in the Vector Count Register (VCR). The values of elements in the destination vector register between the new value of VCR and the vector length are UNPREDICTABLE.

Note

If a large value is specified for the stride.rl operand, there is a chance for integer overflow during calculation of the "tmp <- tmp + stride" step. In this case, the overflow is ignored. For example:

```
                tmp  <-  tmp  + stride

Value of tmp before above step:  FFFFFF00
Value of Stride:                 FFFFFF00

Value of tmp + stride:         1 FFFFFE00
```

Since the overflow is ignored, the new value of tmp is FFFFFE00.

13.10 MISCELLANEOUS INSTRUCTIONS

MFVP Move From Vector Processor

Format:

```
opcode  regnum.rw, dst.wl
```

Operation:

```
CASE regnum OF
   0:   dst <- ZEXT{VCR}
   1:   dst <- ZEXT{VLR}
   2:   dst <- VMR<31:0>
   3:   dst <- VMR<63:32>
   4:   SYNC
        dst <- UNPREDICTABLE
   5:   MSYNC
        dst <- UNPREDICTABLE
  >5:   Reserved
END
```

Move From Vector Processor instructions that specify reserved values of the regnum operand produce UNPREDICTABLE results.

Vector Processor Exceptions:

None

Opcode:

31FD	MFVP	Move From Vector Processor

Description:

This instruction can be used to read the Vector Count, Length, and Mask Registers, and to synchronize a scalar processor with its associated vector processor.

When the scalar processor issues an MFVP instruction to the vector processor, the scalar processor waits for the MFVP result to be written before processing other instructions.

MFVP from VCR or VLR does not read that register until all previous writes to the register are completed. MFVP from VMR<31:0> or VMR<63:32> does not read that longword of VMR until all previous writes to the same longword of VMR are completed; however, this is not true for previous writes to the other longword.

SYNC allows software to ensure that the unreported exceptions of all previously issued vector instructions (including vector memory instructions in asynchronous memory management mode) are detected and reported to the scalar processor before the scalar processor proceeds with further instructions. For more details about SYNC and its exception reporting nature refer to section 14.3.1, Scalar/Vector Instruction Synchronization.

MSYNC allows software to ensure that all previously issued memory instructions of the scalar/vector processor pair are complete before the scalar processor proceeds with further instructions. For more details about MSYNC and its exception reporting nature, refer to section 14.3.2.1, Memory Instruction Synchronization.

The value of the vector control register (VCR, VLR, VMR<31:0>, VMR<63:32>) delivered by an MFVP depends upon the value of certain vector register elements and vector control register bits. Unreported exceptions that occur in the production of these elements and control register bits are reported by the vector processor prior to the completion of the MFVP from the vector control register.

In addition, there are vector register elements and vector control register bits that the value of a vector control register delivered by an MFVP does not depend upon. It is UNPREDICTABLE whether unreported exceptions that occur in the production of these elements and control register bits are reported by the vector processor prior to the completion of the MFVP from the vector control register. Software must not rely upon the reporting of these exceptions prior to the completion of the MFVP for the correctness of program results.

Section 14.1.3.3, Dependences Among Vector Results, gives the necessary rules to determine what vector control register elements and vector control register bits the value of a vector control register delivered by an MFVP depends upon. Examples of MFVP exception reporting using these rules are found in section 14.2.5.

When a vector arithmetic exception or memory management exception (in asynchronous memory management mode) is reported prior to the completion of an MFVP, then:

- o the operation of the MFVP does not complete,
- o no longword result is written to the scalar destination of the MFVP by the scalar processor, and
- o the MFVP itself (rather than the next vector instruction) takes either a vector processor disabled fault or a memory management fault.

After the appropriate fault has been serviced, the MFVP may be returned to through an REI If both exception conditions are encountered by an MFVP, then the MFVP itself takes a vector processor disabled fault. In this case, after the vector processor disabled fault has been serviced, returning to the MFVP instruction will cause the asynchronous memory management exception to be reported.

MTVP Move To Vector Processor

Format:

```
        opcode  regnum.rw, src.rl
```

Operation:

```
        CASE regnum OF
           0:   VCR <- src
           1:   VLR <- src
           2:   VMR<31:0> <- src
           3:   VMR<63:32> <- src
          >3:   Reserved
        END
```

Move To Vector Processor instructions that specify reserved values of the regnum operand produce UNPREDICTABLE results.

Vector Processor Exceptions:

None

Opcode:

A9FD MTVP Move To Vector Processor

Description:

This instruction can be used to write the Vector Count, Length, and Mask Registers.

The new value of VCR, VLR, or VMR does not affect any prior instructions. The new value remains in effect for all subsequent vector instructions executed until a new value is loaded.

VSYNC   Synchronize Vector Memory Access

Format:

```
opcode   regnum.rw
```

Operation:

```
CASE regnum OF
   0:   VSYNC
  >0:   Reserved
END
```

Synchronize Vector Memory Access instructions that specify reserved values of the regnum operand produce UNPREDICTABLE results.

Vector Processor Exceptions:

None

Opcode:

A8FD   VSYNC   Synchronize Vector Memory Access

Description:

The VSYNC instruction can be used to synchronize memory access within the vector processor. The instruction allows software to order the conflicting memory accesses of vector-memory instructions issued after VSYNC with those of vector-memory instructions issued before VSYNC. Specifically, VSYNC forces the access of a memory location by any subsequent vector-memory instruction to wait for (depend upon) the completion of all prior conflicting accesses of that location by previous vector-memory instructions. See section 14.3.4 for more details.

See section 14.3.5, Required Use of Memory Synchronization Instructions, for the conditions when VSYNC is not required before a vector store instruction.

13.11 ASSEMBLER NOTATION

The assembler notation uses a format that is different from the operand specifiers for the vector instructions. The number and order of operands is not the same as the instruction-stream format. For example, vector-to-vector addition is denoted by the assembler as "VVADDL V1, V2, V3" instead of "VVADDL ^X123". The assembler always generates immediate addressing mode (I^#constant) for vector control word operands.

The assembler notation for vector instructions uses opcode qualifiers to select whether vector processor exception conditions are enabled or disabled, and to select the value of cntrl<MTF> in masked, VMERGE, and IOTA operations. The appropriate opcode is followed by a slash (/) and up to two qualifiers specified in any order.

- The qualifier U enables floating underflow. The qualifier V enables integer overflow. Both of these qualifiers set cntrl<EXC>. The default is no vector processor exception conditions are enabled.

- The qualifier 0 denotes masked operation on elements for which the Vector Mask Register bit is 0. The qualifier 1 denotes masked operation on elements for which the Vector Mask Register bit is 1. Both qualifiers set cntrl<MOE>. The default is no masked operations.

- For the VMERGE and IOTA instructions only, the qualifier 0 denotes cntrl<MTF> is 0. The qualifier 1 denotes cntrl<MTF> is 1. Cntrl<MTF> is 1 by default. Cntrl<MOE> is not set in this case.

- For the VLD and VGATH instructions only, the qualifier M indicates modify intent (cntrl<MI> is 1). The default is no modify intent (cntrl<MI> is 0).

Examples:

```
VVADDF/1   V0, V1, V2   ;operates on elements with mask bit set
VVMULD/0   V0, V1, V2   ;operates on elements with mask bit clear
VVADDL/V   V0, V1, V2   ;Enables exception conditions
                         (integer overflow here)
VVSUBG/U0  V0, V1, V2   ;Enables floating underflow and
                        ;operates on elements with mask bit clear

VLDL/M     base,#4,V1   ;Indicates Modify Intent
```

The assembler notation is shown in table 13-8.

Table 13-8: Assembler Notation for Vector Instructions

Assembler Notation		Vector Instruction generated by Assembler		
VLDx	base, stride, Vc	VLDx	cntrl.rw, base.ab, stride.rl	
VGATHx	base, Vb, Vc	VGATHx	cntrl.rw, base.ab	
VSTx	Vc, base, stride	VSTx	cntrl.rw, base.ab, stride.rl	
VSCATx	Vc, base, Vb	VSCATx	cntrl.rw, base.ab	
		[where x = (L,Q)]**		
VVADDx	Va, Vb, Vc	VVADDx	cntrl.rw	
VSADDx	scalar, Vb, Vc	VSADDx	cntrl.rw, addend.ry	
VVSUBx	Va, Vb, Vc	VVSUBx	cntrl.rw	
VSSUBx	scalar, Vb, Vc	VSSUBx	cntrl.rw, min.ry	
VVMULx	Va, Vb, Vc	VVMULx	cntrl.rw	
VSMULx	scalar, Vb, Vc	VSMULx	cntrl.rw, mulr.ry	
VVGTRx	Va, Vb	VVCMPx	cntrl.rw	{cmp_func = GTR}
VSGTRx	src, Vb	VSCMPx	cntrl.rw, src.ry	{cmp_func = GTR}
VVEQLx	Va, Vb	VVCMPx	cntrl.rw	{cmp_func = EQL}
VSEQLx	src, Vb	VSCMPx	cntrl.rw, src.ry	{cmp_func = EQL}
VVLSSx	Va, Vb	VVCMPx	cntrl.rw	{cmp_func = LSS}
VSLSSx	src, Vb	VSCMPx	cntrl.rw, src.ry	{cmp_func = LSS}
VVNEQx	Va, Vb	VVCMPx	cntrl.rw	{cmp_func = NEQ}
VSNEQx	src, Vb	VSCMPx	cntrl.rw, src.ry	{cmp_func = NEQ}
VVGEQx	Va, Vb	VVCMPx	cntrl.rw	{cmp_func = GEQ}
VSGEQx	src, Vb	VSCMPx	cntrl.rw, src.ry	{cmp_func = GEQ}
VVLEQx	Va, Vb	VVCMPx	cntrl.rw	{cmp_func = LEQ}
VSLEQx	src, Vb	VSCMPx	cntrl.rw, src.ry	{cmp_func = LEQ}
		[where <x =(L,F) and Y =(L)> or] [<x =(D,G) and Y =(Q)>]**		
VVDIVx	Va, Vb, Vc	VVDIVx	cntrl.rw	
VSDIVx	scalar, Vb, Vc	VSDIVx	cntrl.rw, divd.ry	
		[where <x =(F) and Y =(L)> or] [<x =(D,G) and Y =(Q)>]**		
VVCVTxy	Vb, Vc	VVCVT	cntrl.rw	{cvt_func = xy}
		[where xy = (FD, FG, GF, DF,] [L{F,D,G}, {F,D,G}L,] [{RF,RD,RG}L,)]**		

** = Note indicates data types used/supported by previous group of instructions.

Table 13-8 Assembler Notation for Vector Instructions (cont)

Assembler Notation	Vector Instruction generated by Assembler
VVBICL Va, Vb, Vc	VVBICL cntrl.rw
VSBICL scalar, Vb, Vc	VSBICL cntrl.rw, src.rl
VVBISL Va, Vb, Vc	VVBISL cntrl.rw
VSBISL scalar, Vb, Vc	VSBISL cntrl.rw, src.rl
VVXORL Va, Vb, Vc	VVXORL cntrl.rw
VSXORL scalar, Vb, Vc	VSXORL cntrl.rw, src.rl
VVSLLL Va, Vb, Vc	VVSLLL cntrl.rw
VSSLLL cnt, Vb, Vc	VSSLLL cntrl.rw, cnt.rl
VVSRLL Va, Vb, Vc	VVSRLL cntrl.rw
VSSRLL cnt, Vb, Vc	VSSRLL cntrl.rw, cnt.rl
IOTA stride, Vc	IOTA cntrl.rw, stride.rl
VVMERGE Va, Vb, Vc	VVMERGE cntrl.rw
VSMERGE src, Vb, Vc	VSMERGE cntrl.rw, src.rq
%% VSMERGEx src, Vb, Vc	VSMERGE cntrl.rw, src.rq
MTVCR src	MTVP S^#0, src.wl
MFVCR dst	MFVP S^#0, dst.wl
MTVLR src	MTVP S^#1, src.wl
MFVLR dst	MFVP S^#1, dst.wl
MTVMRLO src	MTVP S^#2, src.wl
MFVMRLO dst	MFVP S^#2, dst.wl
MTVMRHI src	MTVP S^#3, src.wl
MFVMRHI dst	MFVP S^#3, dst.wl
SYNC dst	MFVP S^#4, dst.wl
MSYNC dst	MFVP S^#5, dst.wl
VSYNC	VSYNC S^#0

%% = For VSMERGEx, x =(F,D,G) and determines the floating point format the assembler will use when src is immediate to encode src as the quadword-length scalar source operand. VSMERGEF creates a scalar source operand whose high-order longword is all zeroes and whose low-order longword is the F_floating point encoding of src. Literal mode is not allowed with the VSMERGEx pseudo-opcodes and attempting to force use of this address mode will result in an assembly warning. Likewise, non-immediate mode addressing is not allowed with VSMERGEF and attempting to force use of this address mode will result in an assembly warning. When src is not immediate or literal, VSMERGEG and VSMERGED assemble exactly as VSMERGE.

/VSMERGEx was added so that the assembler could be instructed to encode an immediate scalar source operand for VSMERGE as either a D_floating datum, a G_floating datum, or a quadword where the high-order longword is all zeros and the low-order longword is a F_floating datum./

Change History:

Revision J. Rich Brunner, December 1989

- o Split sections on execution model and exception reporting into chapter 14.
- o Included chapter into SRM revision J.

Revision 6. Rich Brunner, 11 April 1989.

- o Put in Clarifications to flows in context switching and execution model sections.
- o Put PTE(P) bit back into MM fault parameter.
- o Put in VSMERGEx pseudo-opcodes.
- o MFPR to VPSR needed after MTPR to VPSR to ensure new state of VPSR affects execution of subsequent vector instructions.
- o Revised Vector Processor Disable text
- o Clarified IVO and IMP descriptions
- o For the IOTA vector instruction, bits <63:32> of the destination elements are UNPREDICTABLE.
- o Added Modify-Intent bit with appropriate text
- o Added Clarifications and new subsection on Illegal Vector Opcodes (Matt Kirk wrote it).
- o Changed Format of F & D floating ROPs per ECO
- o Clarified that the destination register mask in VAER does not get written for exceptions that occur from a floating VCMP.
- o New Table For Assembler Notation.
- o No BASE immediate address for vector memory instructions
- o VSCAT writes highest-numbered element to memory location destined by multiple elements.
- o Cntrl<12> reserved for future use by VSCAT for out-of-order writes.
- o VMAC does memory hard-error reporting
- o Waiting on VPSR<BSY> will catch non-memory hard-errors
- o Order of Scalar context flow changed
- o Vector Context flow split off into own section
- o Vector Processor does Modify-Fault if VM option implemented on Scalar.
- o When MME occurs, it is unpredictable whether elements which did not get the MME are written.
- o Zero-strided VST writes highest-numbered element to memory.
- o Completely revised and expanded section on SYNC.
- o Completely revised and expanded section on MSYNC.
- o Completely revised and expanded section on VMAC.
- o Completely revised and expanded section on instruction overlapping.
- o Completely revised and expanded section on chaining.
- o Completely revised and expanded section on register conflict.
- o Added section on dependences between results of vector instructions.
- o Added section on MFVP Exception reporting examples.
- o Completely revised and expanded section on MFVP to match revisions to SYNC, MSYNC and MFVP exception reporting.
- o Completely revised VSYNC description and conditions when it is not required to match dependences concepts.

- SYNC, MSYNC, and VMAC can be interrupted if an implementation chooses to do so.

Revision 5. Cheryl A Wiecek (and Rich Brunner), 10 December 1987.

- Clarified that the exception condition type and destination register number must be recorded in the Vector Arithmetic Exception Register (VAER) for each vector arithmetic exception that occurs.
- At the request of VMS (with no problems for the current implementations), added that MFPR from VMAC has no effect if an implementation supports an optional vector processor, but the vector processor is not installed.
- Removed unsigned vector integer compare (VCMPL) since it won't be used and saves a little pain for implementations. If cntrl<3> is set, the results are UNPREDICTABLE.
- Removed the requirement that the scalar processor must wait for VPSR<BSY> to be clear when MFVP is executed. It was left over from the memory management handling schemes before revision 4.
- Clarified how vector instruction decode proceeds between the scalar and vector processors in the Execution Model section.
- Expanded on how scalar and vector processor context switching works and how vector processor disabled faults are handled. It is in a new subsection of the Execution Model section.
- Allowed multi-mode use of the vector processor and specified the options software has for exception handling across access mode changes. The simplified asynchronous memory mgmt. scheme introduced in revision 4 made this possible.
- Expanded the Synchronization Between the Scalar and Vector Processors section to clarify how MSYNC and VMAC are defined and used, and to include mention of MFVP from vector control registers as a synchronization method. Also reworked the definitions of VMAC in 13.2.3 and MFVP in 13.12, and fixed all uses of "MSYNC" in the spec., to match.
- Added text explaining the code sequences that demonstrate uses for VSYNC.
- Expanded on how scalar/vector memory synchronization must be used in the context of other system components (I/O, other processors, etc.)
- Added that the length of the memory area pointed to by VSAR is implementation-specific in Section 13.2.3 on IPRs.
- Further defined VPSR bits in Table 13.1: VPSR<1,2,3> are RAZ, VPSR<5,6,7,24> are R/W1C, and writing to VPSR<BSY> has no effect.
- Added VPSR<IVO> bit 25 so that Rigel can provide a reason (illegal vector opcode) as the cause of a vector processor disabled fault when this occurs.
- Specified that VPSR<BSY> is clear for suspended instruction execution only in the case of an asynchronous memory management exception.
- Put a pointer to the section on required use of VSYNC in section 13.3.1 (Masked Operations), implementation note 2.
- Expanded on the use vector control word formats fields that produce UNPREDICTABLE results in section 13.3.4 (Vector Control Word Formats).

- o Added text on the use of VMR in section 13.4.3 (Vector Instruction Execution): the VMR value under which an instruction executes must be saved for use during actual execution.
- o Clarified that the P bit can be set in context of scalar processor only in section 13.5.1 (Vector Memory Management Exception Handling)
- o Clarified in section 13.6 (Memory Management) that the vector processor must implement its own memory management registers only as a group and can only load PTEs into its TB when it is executing a vector memory access instruction.
- o Mentioned in the VLD/VGATH instruction descriptions that elements of VLDx and VGATHx may be loaded in any order and more than once. Also, when a vector memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.
- o Eliminated an error in the VCMP instruction description that indicated stored, not true results are used to determine VMR bit values. VxCMP{F,D,G} uses the true result.
- o Expanded text dealing with reserved operands and floating overflow and underflow in the description of VVCVT.
- o Numerous minor corrections and clarifications.

Revision 4. Cheryl A Wiecek, 12 August 1987.

- o Removed synchronous memory management handling using FPD since it was not well-defined and no implementations plan to use it.
- o Redefined clearing VPSR<VEN> while VPSR<BSY> is set and what setting VPSR<RST> does. Made VPSR<MF,PMF,AEX,IMP> write-one-to-clear.
- o Renamed VPSR<EXC> to VPSR<AEX> to distinguish that bit from cntrl<EXC>.
- o Changed VMERGE and IOTA instructions to use cntrl<MTF>.
- o Specified that moving 0 into VTBIA with MTPR invalidates all of the vector translation buffer.
- o Replaced asynchronous memory management handling scheme. This eliminated PVISRs, PVINR and VPSR<CNT> and added VMAC, VSAR and VPSR<STS,RLD,MF,PMF>. The disabling condition terminology was replaced by vector processor exception terminology.
- o Clarified that the vector processor must know the access mode of the scalar processor at the time a vector memory access instruction is issued and use that mode for correct execution of that instruction.
- o Specified a preferred implementation for handling a vector instruction reference to I/O space.
- o Rewrote Execution Model section on Synchronization and added VSYNC instruction.
- o Cleaned up assembler notation section, including allowing opcode qualifiers in any order and noting that immediate mode is always generated for vector control word operands.
- o Added text specifying the result of accesses to nonexistent or illegally-used vector IPRs by MxPR instructions.
- o In the MEMORY MANAGEMENT section, clarified explicit MTPRs to PxBR and PxLR by software due to LDPCTX, clarified why

software must wait for vector processor not busy prior to MTPR to TBIS, TBIA, or VTBIA, and added restriction that vector instructions are not allowed to access page tables.

- Added requirement that for all MFVP cases, scalar processor must wait for vector processor to be idle and then wait for the MFVP result to be written before processing other instructions.
- Added masked operations for VLDx, VSTx, VGATHx, and VSCATx.
- Reassigned vector IPRs from 67-70 to 90-9F (hex).
- For chaining, when an arithmetic exception occurs on the first operation, changed the result written to the source operand of the second operation from reserved operand to default result as defined in the Vector Arithmetic Exceptions section to match PRISM.
- Added note saying that when CVTRGL, CVTRDL, and CVTRFL round, the rounding is done in sign magnitude, before conversion to two's complement.
- Changed the MxVP regnum.rl operand to regnum.rw and changed it from the second to the first specifier in MTVF for consistency with the control word specifier in other vector instructions. The opcode assignment of MTVP changed from 30FD to A9FD, eliminating a format class.
- Made UNPREDICTABLE the value stored in dst for MFVP when regnum exceeds 3.
- Clarified vector processor operation as UNDEFINED when memory management is disabled.
- For vector convert floating to integer instructions, where the source element is a reserved operand, the value written to the destination element changed from encoded reserved operand to UNPREDICTABLE.
- Moved bit indicating vector alignment exception in the fault parameter from bit <31> to bit <3> to keep memory management status in the same byte.
- Added MTVMRLO src and MFVMRHI dst to assembler notation section.
- Numerous minor corrections and clarifications.

Revision 3. Al Thomas, March 1987.

- Numerous clarifications and edits.
- If integer overflow occurs during a vector convert, write the low order 32 bits of the true result to the destination, instead of an encoded reserved operand.
- Changes to P0BR, P0LR, P1BR, and P1LR due to a LDPCTX do not update the copies in the vector processor.
- Remove Integer Divide By Zero Disabling Condition.
- Remove VVDIVL and VSDIVL instructions.
- Add effects of masked operations to instruction pseudo code.
- Change format to match DEC STD 032.
- Rewrite the section on Vector Exceptions.
- Rewrite the section on Execution Model.
- Change MEN to MOE.
- Vector references to I/O space cause UNPREDICTABLE results.
- Emulation of vector instructions is allowed.
- Vector Exceptions changed to Vector Processor Disabling Conditions.
- Vc[i]<63:32> are UNPREDICTABLE after VLDL and VGATHL.
- Reassign IPRs.
- Vc[i]<63:32> are UNPREDICTABLE after CVTxL.
- Explain "Vector Register Conflicts."
- Define "Vector Operate Instructions."
- Dst <- ZEXT{VCR,VLR} after MFVP.
- VMR bits beyond VLR are left unchanged for integer/floating compares.
- Change context of count operand from byte to longword for Vector Shift Logicals.

Revision 2. Dileep Bhandarkar, 14 November 1986.

- Rewrite sections on Execution Model and Exceptions.
- Order of operands for divide and subtract changed to match PRISM, i.e. scalar-vector and scalar/vector.
- Add vector-vector form of shifts.
- Reassign opcodes.
- Remove extra operand for scatter/gather.
- Show cntrl format for each instruction.
- Add processor mode to exception state for load/store.
- Force SYNC to take exception if pending instructions get exception.
- Change bits <63:32> of Vc.
- Instructions that use result of a load as source are allowed to issue before load is guaranteed to complete without exception. VMR is saved in pending instruction status.
- Shorten vector queue.
- Allow synchronous load/stores.
- Assembler mnemonics.

Revision 1. Dileep Bhandarkar, 11 September 1986.

- Initial Distribution.

CHAPTER 14

VAX VECTOR EXECUTION MODEL AND EXCEPTION REPORTING

14.1 EXECUTION MODEL

A typical processor consists of a VAX scalar processor and ıts associated vector processor, which contains vector registers and vector function units. The scalar and vector processors may execute asynchronously. The VAX scalar processor decodes both scalar and vector instructions following the operand specifier evaluation rules stated in Chapter 2, but executes only the scalar instructions. The scalar processor passes the information required to execute a vector instruction to the vector processor. This information may include the vector opcode, scalar source operands, and vector control words. The vector processor performs the required operation, such as loading data from memory, storing data to memory, or manipulating data already loaded into its vector registers.

The scalar processor may decode a vector instruction before checking whether or not the vector processor should receive it. Exceptions on vector instruction operands may occur during this decoding and be taken before the attempt to send the decoded instruction to the vector processor. The scalar processor follows the operation shown below when sending a decoded vector instruction to the vector processor. Recall that because the vector and scalar processors can execute asynchronously, a VPSR state transition may not be seen immediately by the scalar processor.

- If the scalar processor views the vector processor as enabled (the scalar processor sees VPSR<VEN> as set), the decoded vector instruction is sent to the vector processor. The vector processor queues instructions sent by the scalar processor until they can be executed.

- If the scalar processor views the vector processor as disabled (the scalar processor sees VPSR<VEN> as clear), attempting to send the decoded vector instruction to the vector processor results in a vector processor disabled fault.

The following flow details how vector instruction decode proceeds from the scalar processor.

```
DO WHILE (the scalar processor has a decoded vector instruction for
          the vector processor)
  IF (the vector processor is viewed as disabled -- the scalar processor
     sees VPSR<VEN> as clear) THEN
     enter the vector processor disabled fault handler.
  ELSE
     IF (asynchronous memory management handling is implemented
        AND VPSR<PMF> is set) THEN
        enter the memory management exception handler.
        {The vector processor clears VPSR<PMF>.}
     ELSE
        BEGIN
          {If asynchronous memory management handling is
          implemented and VPSR<MF> is set, the vector processor
          clears VPSR<MF>, and retries the faulting memory
          reference before any new vector instructions in the
          queue are executed.}
          IF (the vector processor instruction queue is not full) THEN
             BEGIN
               Send the decoded instruction to the vector processor
               for execution.
               IF (the decoded instruction is a vector memory access
                   instruction AND synchronous memory management
                   handling is implemented) THEN
                  ensure instruction completion without the occurrence
                  of memory management exceptions.
             END
        END
END
```

If asynchronous memory management handling is implemented, and VPSR<MF> is set when the scalar processor sends the vector processor an instruction, the vector processor clears VPSR<MF>, and retries the faulting memory reference before any new vector instructions in the queue are executed.

The VAX scalar processor need not wait for the vector processor to complete its operation before processing other instructions. Thus, the scalar processor could be processing other VAX instructions while the vector processor is performing vector operations. However, if the scalar processor issues an MFVP instruction to the vector processor, the scalar processor must wait for the MFVP result to be written before processing other instructions.

Because the scalar and vector processors may execute asynchronously, it is possible to context switch the scalar processor before the vector processor is idle. Software is responsible for ensuring that scalar and vector memory management remains synchronized, and that all exceptions get reported in the context of the process where they occurred. This is achieved by making sure all vector memory accesses complete, and then disabling the vector processor before any scalar context switch.

\ Software may allow the old process to continue to execute on the vector processor after a scalar context switch. In this case, different processes may be executing on the scalar processor and its associated vector processor. Only when the process running on the scalar processor has a vector instruction to execute will that process be loaded on the vector processor as a result of a vector processor disabled fault. For further details, see section 14.2.4.\

The vector processor may have its own TB and cache and may have separate paths to memory, or it may share these resources with the scalar processor.

14.1.1 Access Mode Restrictions

In general, processes are expected to use the vector processor in only one mode. However, multi-mode use of the vector processor by a process is allowed. Software decides whether to allow vector processor exceptions from vector instructions executed in a previous access mode to be reported in the current mode. The preferred method is to report all vector processor exceptions in the access mode where they occurred. This is achieved by requiring a process that uses the vector processor to execute a SYNC instruction before changing to an access mode where additional vector instructions are executed. The alternative is for software to handle vector processor exceptions resulting from vector instructions executed in a previous mode.

\ In general, a SYNC instruction is not needed upon entry to any called routine that uses the vector processor in the same access mode as the caller. However, if a routine establishes its own condition handler, the preferred method is to execute a SYNC instruction prior to establishing the handler.

In general, a SYNC instruction is not needed prior to calling VMS system services. However, when a process calls a VMS change-mode system service (CMEXEC or CMKRNL) to execute a routine that uses the vector processor in a more-privileged access mode, the preferred method is to execute a SYNC instruction before calling the change-mode system service.

VMS system services that use the vector processor work as follows. (The character string "VP" appears in name of all VMS system services that use the vector processor.)

- o If the service enters a more-privileged access mode, the service executes a SYNC instruction before the mode change because a new condition handler is established in the more-privileged mode. Once in the more-privileged access mode, the service also executes a SYNC instruction after any vector instructions and before the REI that returns to the less-privileged access mode. The service is also required to execute a scalar/vector memory synchronization instruction after any vector memory access instructions in the more-privileged access mode and before the REI that returns to the less-privileged access mode.

- If the service does not change mode, no SYNC is executed by the service. A routine that uses the vector processor may call this service. If a vector instruction executed by the routine results in a vector arithmetic exception, the service gets a vector processor disabled fault on one of the vector instructions issued by the service. VMS gets control and:

  1. Builds a vector arithmetic frame.

  2. Clears VPSR<AEX> and VAER.

  3. Searches for a condition handler to process the exception. If the condition handler unwinds, control is transferred to an earlier procedure in the calling routine. If the condition handler continues, the service's first vector instruction is retried. If no condition handler is found, VMS forces the image to exit. In any case, the vector arithmetic exception is reported, with the PC pointing to the service's vector instruction.

- No service enters a less-privileged access mode, as REI is not allowed to return to a more-privileged access mode. \

For correct access checking of vector memory references, the vector processor must know the access mode in effect when a vector memory access instruction is issued by the scalar processor.

14.1.2 Scalar Context Switching

With the addition of a vector processor, the required steps in performing a scalar context switch change. The following outlines the required method software should use for scalar context switching:

1. Disable the vector processor so that no new vector instructions will be accepted. Writing zero to the VPSR using the MTPR instruction clears VPSR<VEN>, and disables the vector processor without affecting VPSR<31:1>. (See section 14.2.3, Vector Processor Disabled, for more details.)

2. Ensure that no more vector memory reads or writes can occur. Reading the VMAC IPR using the MFPR instruction does the required scalar/vector memory synchronization without any exceptions being reported. Reading VMAC also ensures that all unreported hardware errors encountered by previous vector memory instructions are reported before the MFPR completes. For more information on this function of VMAC, refer to section 14.5, Hardware Errors.

3. Set a software scalar-context-switch flag and perform a normal scalar processor context switch, e.g. SVPCTX, etc., leaving the vector processor state as is.

\ By reading VMAC last, it is likely that most of the previously issued memory instructions will have already completed by the time the VMAC read is issued, thus decreasing the time necessary for the VMAC to complete. \

Although not required by the architecture, software may wait for VPSR<BSY> to be clear after disabling the vector processor when performing a scalar context switch. \VMS will wait for VPSR<BSY> to be clear after disabling the Vector Processor.\ This has a few advantages:

1. the vector processor can not be executing non-memory-access instructions from the previous process while a normal scalar context switch to a new process is being performed -- which may be desirable to an operating system;

2. all unreported hardware errors encountered by previous non-memory-access instructions will be reported by the time the vector processor clears VPSR<BSY> and thus known to software before scalar-context switching continues (refer to section 14.5, Hardware Errors, for more details);

3. the MFPR from VPSR used to read VPSR<BSY> also ensures that the scalar processor views the vector processor as disabled.

If software does not wait for VPSR<BSY> to be clear, it is possible that while a normal scalar context-switch to a new process is being performed, the vector processor may still be executing non-memory-access instructions from the previous process.

The required steps for Vector Context Switching are discussed in section 14.2.4, Handling Disabled Faults and Vector Context Switching.

14.1.3 Overlapped Instruction Execution

To improve performance, the vector processor may overlap the execution of multiple instructions -- that is, execute them concurrently. Further, when no data dependencies are present, it may complete instructions out of order relative to the order in which they were issued. A vector processor implementation can perform overlapped instruction execution by having separate function units for such operations as addition, multiplication, and memory access. Both data-dependent and data-independent instructions can be overlapped; the former by a technique known as chaining, which is described in the next section. In many instances, overlapping allows an operation from one instruction to be performed in any order with respect to an operation of another instruction.

When vector arithmetic exceptions occur during overlapped instruction execution, exception handling software may not see the same instruction state and exception information that would be returned from strictly sequential execution. Most notably, the VAER could indicate the exception conditions and destination registers of a number of vector instructions that were executing concurrently and encountered exceptions. Exception reporting during chained execution is discussed further in the next section.

To ensure correct program results and exception reporting, the architecture does place requirements on the ordering among the operations of one vector instruction and those of another. The primary goal of these requirements is to ensure that the results obtained from both the overlapped and strictly sequential execution of data-dependent instructions are identical. A secondary goal is to establish places within the instruction stream where software is guaranteed to receive the reporting of exceptions from a chain of data-dependent instructions.

In many cases, these requirements ensure the obvious: for example, an output vector register element of one arithmetic instruction must be computed before it can be used as an input element to a subsequent instruction. But, a number of the things ensured are not obvious: for example, an MSYNC must report exceptions encountered in generating a value of VMR that is used in a previously issued masked store instruction.

To precisely define the requirements on the ordering among operations, section 14.1.3.3 discusses the "dependence" among their results (the vector register elements and control register bits produced by the operations).

14.1.3.1 Vector Chaining -

The architecture allows vector chaining, where the results of one vector instruction are forwarded (chained) to another before the input vector of the first instruction has been completely processed. In this way, the execution of data-dependent vector instructions may be overlapped. Thus, chaining is an implementation-dependent feature that is used to improve performance.

With some restrictions stated below, the vector processor may chain a number of instructions. Usually, each instruction is performed by a separate function unit. The number and types of instructions allowed within a chained sequence (often referred to as a "chain") are implementation dependent. Typically, implementations will attempt to chain sequences of two or three instructions such as: operate-operate, operate-store, load-operate, operate-operate-store, and load-operate-store. Load-operate-operate-store may also be possible.

The following is an example of a sequence that an implementation will often chain:

```
        VVADDF  V0, V1, V2
        VVMULF  V2, V3, V4
```

The destination of the VVADDF is a source of the succeeding VVMULF. The VVMULF begins executing when the first sum element of the VVADDF is available.

A number of instructions within a chained sequence can encounter exceptions. For each instruction which encounters an exception, the vector processor records the exception condition type and destination register number in the Vector Arithmetic Exception Register (VAER). When the last instruction within the chain completes, the VAER will show the exception condition type and destination register number of all instructions that encountered exceptions within the chain. Furthermore, when the vector processor disabled fault is finally generated for the exceptions, the VAER may also indicate exception state for instructions issued after the last instruction within the chain. This effect is possible due to the asynchronous exception-reporting nature of the vector processor.

Furthermore, for each instruction which encounters an exception within a chain, the default result, as defined in the Vector Arithmetic Exceptions section (14.2.2), is forwarded as the source operand to the next instruction. This has the effect that default results and exceptions can propagate down through a chain. Note that the default result of one instruction may be overwritten by another instruction before the exception is taken.

Consider the following:

```
        VVADDG V1, V2, V3   ;gets Floating Overflow
        VVGEQG V3, V4       ;gets Floating Reserved Operand
        VVMULG V4, V5, V3   ;overwrites V3
```

For the example above, assume that an exception is taken after the completion of the VVMULG. The VAER will indicate: Floating Overflow and Floating Reserved Operand exception condition types; and V3 as a destination register. However, no default result will be found in the appropriate element of V3 because it has been overwritten by the

VVMULG.

The architecture allows a vector load to be chained into a vector operate instruction provided the operate instruction can be suspended and resumed to produce the correct result if the vector load gets a memory management exception. Consider this example:

```
                VLDL    A, #4, V0
                VVADDF  V0, V1, V1
```

In synchronous memory management mode, the VVADDF cannot be chained into the VLDL until the VLDL is ensured to complete without a memory management exception. This is because the scalar processor is not even allowed to issue the VVADDF to the vector processor until the memory management checks for the VLDL have been completed. In asynchronous memory management mode, the VVADDF may be chained into the VLDL prior to the completion of memory management exception checking. This is possible because a memory management exception in asynchronous memory management mode provides sufficient state to restart both the VLDL and the VVADDF when the memory management exception is corrected.

The architecture allows a vector operate instruction to be chained into a store instruction. If the vector operate instruction encounters an arithmetic exception, the exception condition type and destination register number are recorded in the Vector Arithmetic Exception Register (VAER). The default result generated by that instruction (in some cases an encoded reserved operand) may be written to memory by the store instruction before the exception is reported.

\ The VAX scalar architecture never causes a reserved operand to be stored in memory as a result. \

14.1.3.2 Register Conflict -

When overlapping the execution of instructions, the vector processor must deal with register conflict. This occurs when one instruction is intending to write a register while previously issued instructions are reading from that register. The following is an example of vector register conflict:

```
VVADDF  V1, V2, V3
VVMULF  V4, V5, V1
```

In the example, the VVADDF and VVMULF cannot both begin execution simultaneously because the elements of V1 generated by the VVMULF would overwrite the original elements of V1 required as input by the VVADDF. However, a vector processor implementation can still overlap the execution of these two instructions in a number of ways. One way would be by not starting the VVMULF until the first element of V1 has been read by the VVADDF. In this manner, as the VVADDF reads the next elements from V1 and V2, the VVMULF writes its product into the previous element of V1. This process continues until all the elements have been processed by both instructions. The VVADDF will finish execution while the VVMULF still has at least one product to store.

In the case of the Vector Mask Register (VMR), the vector processor ensures that register conflict does not occur. This is often accomplished by making a copy of the VMR value under which a pending vector instruction is to execute, and using this copy when execution begins. This allows the vector processor to begin executing an instruction that writes VMR before it completes prior instructions that read VMR.

14.1.3.3 Dependences Among Vector Results -

In order to achieve correct results and exception reporting during overlapped execution, the vector processor must maintain certain dependences among the register elements and control register bits produced by various vector instructions. Because of the vector processor's asynchronous exception reporting nature and out-of-order completion of instructions, these dependences differ from those ensured by the VAX scalar processor. In addition, these dependences are at the level of vector register elements and vector control register bits; rather than at the level of vector registers and vector control registers.

Among other things, these dependences determine the exception reporting nature of the MFVP instruction. The value of the vector control register (VCR, VLR, VMR<31:0>, VMR<63:32>) delivered by an MFVP depends upon the value of certain vector register elements and vector control register bits. Unreported exceptions that occur in the production of these elements and control register bits are reported by the vector processor prior to the completion of the MFVP from the vector control register.

The dependences are expressed formally for the various classes of vector instructions by the tables of pseudo-code in this section. These are the only dependences that software should rely upon the vector processor to ensure.

A vector processor implementation is allowed to ensure more than just these dependences providing that this larger set of dependences yields correct results and exception reporting. \ For example consider the following:

```
        VVADDL V5, V6, V7
        VVADDL V1, V2, V3
```

For any value of "i" in the range of 0 to VLR-1 inclusive: besides ensuring the dependence of V3[i] on V1[i], V2[i], and VLR in the second VVADDL, an implementation is allowed to force a dependence of V3[i] on the completion of the first VVADDL due to the availability of the ADD functional unit and other hardware resources.\

NOTE

Note the following for the below tables. Implicit in statements of the form: "result DEPENDS on B" is the requirement that the result depends only on the value of "B" generated by the most immediate previously-issued instruction relative to the result's own generating instruction. For example, in the following, the V3 produced by the VVMULF has the dependence: "V3[i] DEPENDS on V7[i]". This means that the value of V3[i] produced by the VVMULF depends only on the value of V7[i] produced by the VVADDF.

```
VVSUBF V5, V6, V7
VVADDF V1, V2, V7
VVMULF V7, V7, V3
VVDIVF V1, V4, V7
```

Table 14-1: Dependences for Vector Operate Instructions

VVADDx, VSADDx, VVSUBx, VSSUBx, VVMULx, VSMULx, VVDIVx, VSDIVx, VVCVTxy, VVBICL, VSBICL, VVBISL, VSBISL, VVXORL, VSXORL, VVSLLL, VSSLLL, VVSRLL, VSSRLL:

```
for i = 0 to VLR-1
    begin
    Vc[i] DEPENDS on VLR;
    if {MOE EQL 1} then Vc[i] DEPENDS on VMR<i>;
       if ( {MOE EQL 1} AND {VMR<i> EQL MTF} ) OR {MOE EQL 0} then
          begin
          Vc[i] DEPENDS on Vb[i];
          if {Vector-Vector Operation} AND NOT {VVCVTxy} then
             Vc[i] DEPENDS on Va[i];
          end;
    end;
```

Table 14-2: Dependences for Vector Load and Gather Instructions

```
========================================================================
VLDx, VGATHx:
------------------------------------------------------------------------
for i = 0 to VLR-1
    begin
    Vc[i] DEPENDS on VLR;
    if {MOE EQL 1} then Vc[i] DEPENDS on VMR<i>;
    if ( {MOE EQL 1} AND {VMR<i> EQL MTF} ) OR {MOE EQL 0} then
        begin
        if VGATH then
           begin
           Vc[i] DEPENDS on Vb[i];
           k = BASE + Vb[i];
           end
        else
           k = BASE + i * STRIDE;
        Vc[i] DEPENDS on LOAD_COMPLETED(k);
        end;
    end;
------------------------------------------------------------------------
```

Table 14-3: Dependences for Vector Store and Scatter Instructions

```
========================================================================
VSTx, VSCATx
------------------------------------------------------------------------
j = 0;
for i = 0 to VLR-1
    begin
    if ( {MOE EQL 1} AND {VMR<i> EQL MTF} ) OR {MOE EQL 0} then
        begin
        if {MOE EQL 1} then ELEMENT_STORED[j] depends on VMR<i>;
        ELEMENT_STORED[j] DEPENDS on Vc[i];
        ELEMENT_STORED[j] DEPENDS on VLR;
        if VSCAT then
           begin
           ELEMENT_STORED[j] DEPENDS on Vb[i];
           k = BASE + Vb[i];
           end
        else
           k = BASE + i * STRIDE;
        STORE_COMPLETED(k) DEPENDS on ELEMENT_STORED[j];
        j = j+1;
        end;
    end;
------------------------------------------------------------------------
```

Table 14-4: Dependences for Vector Compare Instructions

```
VVCMPx, VSCMPx
```

```
for i = 0 to VLR-1
    begin
    VMR<i> DEPENDS on VLR;
    if {MOE EQL 1} then VMR<i> DEPENDS on VMR<i>
    if ( {MOE EQL 1} AND {VMR<i> EQL MTF} ) OR {MOE EQL 0} then
        begin
        VMR<i> DEPENDS on Vb[i];
        if VVCMP then VMR<i> DEPENDS on Va[i];
        end;
    end;
```

Table 14-5: Dependences for Vector MERGE Instructions

```
VVMERGE, VSMERGE
```

```
for i = 0 to VLR-1
    begin
    Vc[i] DEPENDS on VLR;
    Vc[i] DEPENDS on VMR<i>;
    if {VMR<i> EQL MTF} then
       begin
       if VVMERGE then Vc[i] DEPENDS on Va[i];
       end
    else
        Vc[i] DEPENDS on Vb[i];
    end;
```

Table 14-6: Dependences for IOTA Instruction

```
IOTA
```

```
j = 0;
for i = 0 to VLR-1
    begin
    Vc[j] DEPENDS on VLR;
    if {VMR<i> EQL MTF} then
        begin
        Vc[j] DEPENDS on VMR<0..i>;
        j = j+1;
        end;
    end;
VCR DEPENDS on VMR<0..VLR-1>;
```

Table 14-7: Dependences for MFVP Instructions

MSYNC: DEPENDS on these:

- All STORE_COMPLETED(x) of previously-issued VST & VSCAT
- All LOAD_COMPLETED(X) of previously-issued VLD & VGATH

SYNC: DEPENDS on the vector register elements and vector control register bits produced and stored by all previous vector instructions

MFVMRLO: DEPENDS on VMR<0..31>

MFVMRHI: DEPENDS on VMR<32..63>

MFVCR: DEPENDS on VCR
MFVLR: DEPENDS on VLR

Table 14-8: Miscellaneous Dependences

VSYNC: Depends on nothing, but for each memory location, x, forces:

All subsequent LOAD_COMPLETED(x) and STORE_COMPLETED(x) to DEPEND on all previous LOAD_COMPLETED(x) and STORE_COMPLETED(x).

MTVP: DEPENDS on Nothing

Value of A Memory Location:

The value of a memory location DEPENDS on nothing and is not DEPENDED on by any vector instruction

Transitive Dependence:

if {a DEPENDS on b} AND {b DEPENDS on c} then a DEPENDS on c

14.2 VECTOR PROCESSOR EXCEPTIONS

There are two major classes of vector processor exceptions:

1. Vector Memory Management
   - Access Control Violation
     - Vector Access Control Violation
     - Vector Alignment
     - Vector I/O Space Reference
   - Translation Not Valid
   - Modify
2. Vector Arithmetic
   - Floating Underflow
   - Floating Divide by Zero
   - Floating Reserved Operand
   - Floating Overflow
   - Integer Overflow

   Floating underflow and integer overflow can be disabled on a per-instruction basis by clearing cntrl<EXC>.

Vector processor arithmetic exceptions cause the vector processor to disable itself (see section 14.2.3, Vector Processor Disabled). The vector processor does not disable itself for vector processor memory management exceptions.

14.2.1 Vector Memory Management Exception Handling

Vector processor memory management exceptions are taken through the SCB vector for their scalar counterparts. Figure 14-1 illustrates the memory management fault stack frame that contains the memory management fault parameter.

- The length (L) bit, the PTE reference (P) bit, and the modify or write intent (M) bit are defined in Chapter 4, Memory Management. Vector processor memory management exceptions set these bits in the same way as required for scalar memory management exceptions.

- The vector alignment exception (VAL) bit must be set when an access control violation has occurred due to a vector element not being properly aligned in memory.
- The vector I/O space reference (VIO) bit is set by some implementations to indicate that an access control violation has occurred due to a vector instruction reference to I/O space.
- The vector asynchronous memory management exception (VAS) bit must be set to indicate that a vector processor memory management exception has occurred when the asynchronous memory management scheme described below is implemented.

If more than one kind of memory management exception could occur on a reference to a single page, then access control violation takes precedence over both translation not valid and modify. If more than one kind of access control violation could occur, the precedence of vector access control violation, vector alignment exception, and vector I/O space reference is UNPREDICTABLE.

The architecture allows an implementation to choose one of two methods for dealing with vector processor memory management exceptions. The two methods are as follows:

- Synchronous memory management handling and restart from the beginning.
- Asynchronous memory management handling and store/reload implementation-specific state using VSAR.

With the synchronous method, no new instructions are processed by the vector or the scalar processor until the vector memory access instruction is guaranteed to complete without incurring memory management exceptions. In such an implementation, the vector memory access instruction is backed up when a memory management exception occurs and a normal VAX memory management (access control violation, translation not valid, modify) fault taken with the PC pointing to the faulting vector memory access instruction. If the synchronous method is implemented, VSAR is omitted. After fixing the vector processor memory management exception, software may REI back to the faulting vector instruction. Alternately, software may context switch to another process. For further details on this, see section 14.2.4.

With the asynchronous method, vector memory management exceptions set VPSR<PMF> and VPSR<MF>. The vector processor does not inform the scalar processor of the exception condition; the scalar processor continues processing instructions. All pending vector instructions that have started execution are allowed to complete if their source data is valid. The scalar processor is notified of exception condition(s) when it sends the next vector instruction to the vector processor and a normal VAX memory management fault is taken. The saved PC points to this instruction, which is not the vector memory access instruction that incurred the memory management exception. At this point, the vector processor clears VPSR<PMF>. After fixing the vector processor memory management exception, software may allow the current scalar/vector process to continue. Before vector processor instruction execution resumes using state that already exists in the vector processor, the vector processor clears VPSR<MF> and the faulting memory reference is retried. Alternately, software may context switch to another process. For further details on this, see section 14.2.4.

When a vector processor memory management exception is encountered by a VLD or VGATH instruction, the contents of the destination vector register elements are UNPREDICTABLE. When a vector processor memory management exception is encountered by a VSTL or VSCAT instruction, it is UNPREDICTABLE whether the vector processor writes any result location for which an exception did not occur. In either case, if the fault condition can be eliminated by software and the instruction restarted then the vector processor will ensure that all destination register elements or result locations are written.

```
 3
 1                                                 6 5 4 3 2 1 0
+---------------------------------------------------+-+-+-+-+-+-+
|                                                   |V|V|V| | | |
|                           0                       |A|I|A|M|P|L|  :(SP)
|                                                   |S|O|L| | | |
+---------------------------------------------------+-+-+-+-+-+-+
|           some virtual address in the faulting page           |
+---------------------------------------------------------------+
|                      PC at time fault taken                   |
+---------------------------------------------------------------+
|                      PSL at time fault taken                  |
+---------------------------------------------------------------+
```

Figure 14-1 Memory Management Fault Stack Frame

(as sent by the vector processor)

14.2.2 Vector Arithmetic Exceptions

Vector arithmetic exceptions are not reported like their scalar counterparts; there is no vector arithmetic exception vector in the SCB. When a vector arithmetic exception occurs, it is reported by a vector processor disabled fault which occurs sometime after the arithmetic exception was recognized. The PC reported by the disabled fault points to some vector instruction issued after the vector instruction which encountered the arithmetic exception; this PC value need not point to the next vector instruction issued after the one that got the exception and in practice seldom does. Thus vector arithmetic exception reporting is not "precise" -- the PC value reported when the disabled fault occurs is not necessarily the PC value of the vector instruction which encountered the arithmetic exception.

Vector operate instructions are always executed to completion, even if a vector arithmetic exception occurs. If an exception occurs, a default result is written. The default result is:

- The low-order 32 bits of the true result for integer overflow.
- Zero for floating underflow if exceptions are disabled.
- An encoded reserved operand for floating divide by zero, floating overflow, reserved operand, and enabled floating underflow. (See Section 13.8.1.) For vector convert instructions that convert floating-point data to integer data, where the source element is a reserved operand, the value written to the destination element is UNPREDICTABLE.

The exception condition type and destination register number are always recorded in the Vector Arithmetic Exception Register (VAER) when a vector arithmetic exception occurs. Refer to section 13.2.3, Internal Processor Registers for more information.

14.2.3 Vector Processor Disabled

As a result of error conditions or software control, the vector processor signals the scalar processor not to issue any more vector instructions. The vector processor is disabled when this signal is generated and its state reflected in VPSR<VEN>. Because the scalar and vector processors can execute asynchronously, the scalar processor may not receive this signal immediately. As a result, the scalar processor may continue to view the vector processor as enabled and send it vector instructions. Once the scalar processor receives this signal, it will view the vector processor as disabled and not send it any more vector instructions (including MFVP/MTVP). While the vector processor is disabled, and in the absence of hardware errors, it will complete all pending instructions in its instruction queue including those sent by the scalar processor after the vector processor became disabled.

The vector processor can either disable itself or be disabled by software. The following error conditions cause the vector processor to disable itself:

- o Vector Arithmetic Exception (flagged by VPSR<AEX>)
- o Hardware error (flagged by VPSR<IMP> in some implementations)
- o On some implementations, Receipt of an Illegal Vector Opcode (flagged by VPSR<IVO>)

In these cases, the vector processor clears VPSR<VEN> and flags the error condition by setting the appropriate bit in VPSR. (See Table 13-1.)

Software disables the vector processor by writing a zero into VPSR<VEN> using an MTPR instruction. Once the vector processor is disabled, only software can enable it. The software does this by writing a one to VPSR<VEN> using an MTPR. Recall that after performing an MTPR to VPSR, software must then issue an MFPR from VPSR to ensure that the new state of VPSR will affect the execution of subsequently issued vector instructions. The MFPR will not complete in this case until the new state of the vector processor becomes visible to the scalar processor.

When the vector processor disables itself due to a hardware error, it is implementation-dependent whether the vector processor completes any pending vector instruction. However, in this case, the vector processor ensures when it is re-enabled that all uncompleted instructions have been flushed from the instruction queue. \This requirement is necessary because the vector processor can not be context switched with pending instructions -- there is no architecturally defined way for software to extract the internal state associated with pending instructions so as to restart them later. \

If the scalar processor attempts to issue a vector instruction after it views the vector processor as disabled, then a vector processor disabled fault occurs. This fault uses the vector at SCB offset 68 (hex) and pushes the values of the PC and PSL at the time the fault was taken on to the stack. The exception handling software (running on the scalar processor) can then read the vector Internal Processor Registers with MFPR instructions to determine what exception conditions are recorded in the vector processor and if the vector processor is still busy processing other unfinished instructions.

Once the scalar processor views the vector processor as disabled, the only operations that can be issued to the vector processor are MTPR and MFPR to/from the vector IPRs.

14.2.4 Handling Disabled Faults and Vector Context Switching

The following flow outlines the required steps for handling a vector processor disabled fault.

If the new process executing on the scalar processor has a vector instruction to execute, saving and restoring the state of the vector processor -- that is, vector context switching -- is done as part of handling a subsequent vector processor disabled fault.

If a vector processor disabled fault occurs and the current scalar process is also the current vector process, then software must:

1. Obtain the vector processor status by reading the VPSR using the MFPR instruction.

2. Perform the following checks to see if any of these conditions caused the vector processor to be disabled. If any of these conditions exist, a decision to not continue this flow may occur.

   o If VPSR<IVO> is set, then write one to clear VPSR<IVO> using the MTPR instruction, and report an illegal vector opcode error.

   o If VPSR<IMP> is set, then write one to clear VPSR<IMP> using the MTPR instruction, and report an implementation-specific error.

   o If VPSR<AEX> is set, then write one to clear VPSR<AEX> using the MTPR instruction, and enter the vector arithmetic exception handler with information in VAER.

3. If the software scalar-context-switch flag is set, indicating that a scalar context switch has been done, then:

   o Make sure the vector processor has access to correct P0LR, P0BR, P1LR, and P1BR values.

   o If any vector translation buffer needs to be invalidated, then write zero into the VTBIA IPR using the MTPR instruction. Vector translation buffer flushing is required if the process was swapped out and the mapping change has not yet been made known to the vector translation buffer.

   o Clear the software scalar-context-switch flag.

4. Enable the vector processor by writing one to VPSR<VEN> using the MTPR instruction. Ensure the new state of the vector processor becomes visible to the scalar processor by reading VPSR with the MFPR instruction.

5. REI to retry the vector instruction at the time of the vector processor disabled fault. If there is an asynchronous memory management exception pending, it is taken when that vector instruction is re-issued to the vector processor.

If a vector processor disabled fault occurs and the current scalar process is not the current vector process, then software must:

1. Check if there is a current vector process. If there is one, then:

   a. Wait for VPSR<BSY> to be clear using the MFPR instruction. \ Since VMS waits for VPSR<BSY> to be clear before the scalar context switch, VPSR<BSY> is already clear at this point. \

   b. Perform the following check to see if this condition caused the vector processor to be disabled. If this condition exists, a decision to not continue this flow may occur.

      o If VPSR<IMP> is set, then report an implementation-specific error.

   c. If VPSR<IVO> is set, then set a software IVO flag for this process. The illegal vector opcode error is handled when this process next tries to execute in the vector processor.

   d. If VPSR<AEX> is set, then set a software AEX flag for this process, and save vector arithmetic exception state from VAER using the MFPR instruction. Any vector arithmetic exception conditions are handled when this process next tries to execute in the vector processor.

   e. At this point there can't be a synchronous memory management exception pending. But, if asynchronous memory management handling is implemented, there may be an asynchronous memory management exception pending. Because scalar/vector memory synchronization was required before scalar context switching, all such pending exceptions are known at this time. So, if VPSR<PMF> is set, then:

      o Set a software asynch-memory-exception-pending flag for this process.

      o Store implementation-specific vector state in memory starting at the address in VSAR by writing one to VPSR<STS> using the MTPR instruction.

   f. Reset the vector processor state to clear VAER and VPSR, and enable the vector processor. Writing a one to both VPSR<RST> and VPSR<VEN> using the same MTPR instruction accomplishes this. Ensure the new state of the vector processor becomes visible to the scalar processor by reading VPSR with the MFPR instruction.

g. Store the current vector (V0 - V15) and vector control (VLR, VMR, and VCR) register values using VST and MFVP instructions.

h. Read the VMAC IPR using the MFPR instruction. This ensures scalar/vector memory synchronization and that all hardware errors encountered by previous vector memory instructions have been reported.

2. Make the current scalar process also the current vector process.

3. Clear the software scalar-context-switch flag.

4. Make sure the vector processor has access to correct P0LR, P0BR, P1LR, and P1BR values, and invalidate any vector translation buffer by writing zero to the VTBIA IPR using the MTPR instruction.

5. Load the saved vector (V0 - V15) and vector control (VLR, VMR, and VCR) register values using VLD and MTVP instructions.

6. If the software IMP, IVO, or AEX flags for this process are set, then:

   o Disable the vector processor by writing zero to VPSR<VEN> using the MTPR instruction. Ensure the new state of the vector processor becomes visible to the scalar processor by reading VPSR with the MFPR instruction.

   o If set, clear the software IMP flag for this process and finish handling the implementation-specific error. A decision to not continue this flow may occur.

   o If set, clear the software IVO flag for this process and report an illegal vector opcode error occurred. A decision to not continue this flow may occur.

   o If set, clear the software AEX flag for this process and enter the vector arithmetic exception handler with saved VAER state. A decision to not continue this flow may occur.

7. If the software async-memory-exception-pending flag for this process is set, then:

   o Clear the software async-memory-exception-pending flag for this process.

   o Send the vector processor the memory address that points to implementation-specific vector state for this process by writing VSAR using the MTPR instruction.

- Reload the implementation-specific vector state for this process and leave the vector processor enabled by writing one to both VPSR<RLD> and VPSR<VEN> using the same MTPR instruction. From this state, the vector processor determines if VPSR<PMF> and/or VPSR<MF> need to be set, and does it. Ensure the new state of the vector processor becomes visible to the scalar processor by reading VPSR with the MFPR instruction.

8. REI to retry the vector instruction at the time of the vector processor disabled fault. If there is an asynchronous memory management exception pending, it is taken when that vector instruction is re-issued to the vector processor.

14.2.5 MFVP Exception reporting examples

This section gives examples of MFVP exception reporting that are ensured by the vector processor. The rules used to determine the correct result for each example are found in: the tables of dependences found in section 14.1.3.3, the description of MSYNC in section 14.3.2.1, and the description of MFVP in section 13.10.

These examples illustrate which exceptions are ensured by the vector processor to always cause MSYNC to fault:

1.

```
VVMULF    V1, V1, V2
VVADDF    V3, V2, V3
MTVLR     #1
VSTL      V2, A,  #4
VVCVTFD   V2, V3
MSYNC     R0
```

The MSYNC faults if exceptions occur in the production of V2[0] by the VVMULF or in the storage of V2[0] by the VSTL. MSYNC need not fault if exceptions occur in the production of: V2[1..VLR-1] by the VVMULF, V3[0..VLR-1] by the VVADDF, or V3[0..VLR-1] by the VVCVTFD.

2.

```
VVADDF    V1, V1, V0
VLDL      A,  #4, V0
MSYNC     R0
```

The MSYNC faults if exceptions occur in the loading of V0[0..VLR-1] from memory. MSYNC need not fault if exceptions occur in the production of V0[0..VLR-1] by the VVADDF.

3.

```
VVADDF    V1, V1, V2
VLDL      A,  #4, V1
MSYNC     R0
```

The MSYNC faults if exceptions occur in the loading of V1[0..VLR-1] from memory. MSYNC need not fault if exceptions occur in the production of V2[0..VLR-1] by the VVADDF.

4.

```
VVMULF    V1, V1, V2
VVGTRF    V2, V3
VSTL/1    V0, A,  #4
MSYNC     R0
```

The MSYNC faults if exceptions occur: in the production of V2[0..VLR-1] by the VVMULF, in the production of VMR<0..VLR-1> by the VVGTRF, or in the storage by the VSTL/1 of elements of V0 for which the corresponding VMR bit is one.

These examples illustrate which exceptions the vector processor will report prior to the completion of an MFVP from a vector control register:

1.

```
VLDL       A, #4, V1
VVMULF     V1, V1, V2
MTVLR      #1
VVGTRF     V2, V3
MFVMRHI    R1
MFVMRLO    R2
```

Unreported exceptions that occur: in the loading of V1[0] from memory by the VLDL, in the production of V2[0] by the VVMULF, and VMR<0> by the VVGTRF are reported by the vector processor prior to the completion of the MFVMRLO. The vector processor need not at that time report any exceptions that occur: in the loading of V1[1..63] from memory by the VLDL or in the production of V2[1..63] by the VVMULF. Note that the vector processor need not report any exceptions before completing MFVMRHI.

2.

```
VVGTRF     V0, V1
MTVMRLO    #patt
MFVMRLO    R1
```

For any value of "i" in the range of 0 to 31 inclusive: The value of VMR<i> delivered by MFVMRLO only depends on the value placed into VMR<i> by the MTVMRLO. As a result, the vector processor need not report exceptions that occur in the production of VMR by the VVGTRF prior to completing the MFVMRLO.

3.

```
VVMULF/1   V1, V1, V2
MTVMRLO    #patt
MFVMRLO    R1
```

For any value of "i" in the range of 0 to 31 inclusive: The value of VMR<i> delivered by MFVMRLO only depends on the value placed into VMR<i> by the MTVMRLO. As a result, the vector processor need not report exceptions that occur in the production of V2[0..VLR-1] by the VVMULF/1 prior to completing the MFVMRLO.

4.

```
MTVLR      #64
VVMULF     V0, V0, V2
VVGTRF     V0, V2
MTVLR      #32
IOTA       #str, V4
MFVCR      R1
```

Prior to the completion of the MFVCR, the vector processor must report any exceptions that occurred in the production of: V2[0..31] by the VVMULF and VMR<0..31> by the VVGTRF. Note that VCR produced by an IOTA depends only on VMR<0..VLR-1>. Recall that no exceptions can occur in the production of V4[0..VCR-1] by IOTA.

5.

```
                MTVLR      #64
                VLDL       A, #4, V2
                VVGTRF     V0, V1
                VSGTRF/1   #3.0, V2
                MFVMRLO    R1
```

For any value of "i" in the range of 0 to 31 inclusive: prior to the completion of the MFVMRLO, the vector processor must report any exceptions that occurred: in the loading of V2[i] from memory for which V0[i] is greater than V1[i], in the production of VMR<0..31> by the VVGTRF, and in the production of VMR<0..31> by the VSGTRF/1.

6.

```
                VVMULF     V1, V1, V1
                VSTL       V1, base, #str
                MTVMRLO    base
                MFVMRLO    R1
```

In this example, the value of VMR<31:0> delivered by MFVMRLO only depends on the value placed into VMR<31:0> by the MTVMRLO -- whether this value is V1[0] or the previous value of the location is UNPREDICTABLE. As a result, the vector processor need not report exceptions that occur in the production of V1 by the VVMULF or in the storage of V1 by the VSTL.

14.3 SYNCHRONIZATION

For most cases, it is desirable for the vector processor to operate concurrently with the scalar processor so as to achieve good performance. However, there are cases where the operation of the vector and scalar processors must be synchronized to ensure correct program results. Rather than forcing the vector processor to detect and automatically provide synchronization in these cases, the architecture provides software with special instructions to accomplish the synchronization. These instructions synchronize:

1. exception reporting between the vector and scalar processors;
2. memory accesses between the scalar and vector processors; and
3. memory accesses between multiple load/store units of the vector processor.

Software must determine when to use these synchronization instructions to ensure correct results.

The following sections describe the synchronization instructions.

14.3.1 Scalar/Vector Instruction Synchronization (SYNC)

A mechanism for scalar/vector instruction synchronization between the scalar and vector processors is provided by SYNC, which is implemented by the MFVP instruction. SYNC allows software to ensure that the exceptions of previously issued vector instructions are reported before the scalar processor proceeds with the next instruction. SYNC detects both arithmetic exceptions and asynchronous memory management exceptions and reports these exceptions by taking the appropriate VAX instruction fault. Once it issues the SYNC, the scalar processor executes no further instructions until the SYNC completes or faults.

In beginning the execution of SYNC, the vector processor determines if any previously-issued vector instruction has encountered exceptions which have yet to be reported to the scalar processor. If so, the SYNC is faulted; otherwise, the vector processor waits for either of the following conditions to be true:

1. a pending or currently executing vector instruction encounters an exception -- in which case the SYNC faults; or
2. the vector processor determines that all pending and currently-executing vector instructions (including memory instructions in asynchronous memory management mode) will execute to completion without encountering vector exceptions. In that case the SYNC completes.

When SYNC completes, a longword value (which is UNPREDICTABLE) is returned to the scalar processor. The scalar processor writes the longword value to the scalar destination of the MFVP and then proceeds to execute the next instruction. If the scalar destination is in memory, it is UNPREDICTABLE whether the new value of the destination becomes visible to the vector processor until scalar/vector memory synchronization is performed.

When SYNC faults, it is not completed by the vector processor and the scalar processor does not write a longword value to the scalar destination of the MFVP. Also, depending on the exception condition encountered, the SYNC itself takes either a vector processor disabled fault or memory management fault. If both faults are encountered while the vector processor is performing SYNC, then the SYNC itself takes a vector processor disabled fault. Note that it is UNPREDICTABLE whether the vector processor is idle when the fault is generated. After the appropriate fault has been serviced, the SYNC may be returned to through an REI.

\If the time necessary to execute a SYNC is long, an implementation may allow the execution of the SYNC to be interrupted so as to service an interrupt request. In this case, the SYNC must be restartable after the interrupt is serviced. Allowing SYNC to be interrupted is not a requirement; however, implementations should consider this option if the execution of SYNC could significantly delay the servicing of device interrupts. At the moment, no current implementation is planning on doing this.\

SYNC only affects the scalar/vector processor pair that executed it. It has no effect on other processors in a multiprocessor system.

14.3.2 Scalar/ Vector Memory Synchronization

Scalar/vector memory synchronization allows software to ensure that the memory activity of the scalar/vector processor pair has ceased and the resultant memory writes have been made visible to each processor in the pair before the pair's scalar processor proceeds with the next instruction. Two ways are provided to ensure scalar/vector memory synchronization: using MSYNC, which is implemented by the MFVP instruction, and using the MFPR instruction to read the VMAC (Vector Memory Activity Check) Internal Processor Register. The next two sections discuss MSYNC and VMAC in detail.

Scalar/vector memory synchronization does not mean that previously issued vector memory instructions have completed; it only means that the vector and scalar processor are no longer performing memory operations. While both VMAC and MSYNC provide scalar/vector memory synchronization, MSYNC performs significantly more than just that function. In addition, VMAC and MSYNC differ in their exception behavior.

Note that scalar/vector memory synchronization only affects the scalar/vector processor pair that executed it. It has no effect on other processors in a multiprocessor system. Scalar/vector memory synchronization does not ensure that the writes made by one scalar/vector pair are visible to any other scalar or vector processor.

Software can make data visible and shared between a scalar/vector pair and other scalar and vector processors by using the mechanisms described in section 7.1, Memory, Multiprocessing, and Interprocessor Communication. Software must first make a memory write by the vector processor visible to its associated scalar processor through scalar/vector memory synchronization before making the write visible to other processors. Without performing this scalar/vector memory synchronization, it is UNPREDICTABLE whether the vector memory write will be made visible to other processors even by the mechanisms described in section 7.1.

Lastly, waiting for VPSR<BSY> to be clear does not guarantee that a vector write is visible to the scalar processor.

14.3.2.1 Memory Instruction Synchronization (MSYNC) -

While MSYNC performs scalar/vector memory synchronization, it does more than that. MSYNC allows software to ensure that all previously issued memory instructions of the scalar/vector processor pair are complete and their results made visible before the scalar processor proceeds with the next instruction.

MSYNC is implemented through the non-privileged MFVP instruction. Arithmetic and asynchronous memory management exceptions encountered by previous vector instructions can cause MSYNC to fault.

Once it issues MSYNC, the scalar processor executes no further instructions until MSYNC completes or faults. MSYNC completes when:

1. all previously issued scalar and vector memory instructions have completed;

2. all resultant memory writes (scalar writes and vector stores) have been made visible to both the scalar and vector processor; and

3. no exception that should cause MSYNC to fault (as described below) has occurred.

MSYNC faults when:

1. any unreported exception has occurred in the production or storage of any result (vector register element or vector control register bit) that MSYNC depends upon. Such results include all elements loaded or stored by a previously issued vector memory instruction as well as any element or control register bit that these elements depend upon.

It is UNPREDICTABLE whether MSYNC faults due to exceptions that occur in the production and storage of results (vector register elements and vector control register bits) that MSYNC does not depend upon. Software should not rely on such exceptions being reported by MSYNC for program correctness.

When MSYNC completes, a longword value (which is UNPREDICTABLE) is returned to the scalar processor, which writes it to the scalar destination of the MFVP. The scalar processor then proceeds to execute the next instruction. If the scalar destination is in memory, it is UNPREDICTABLE whether the new value of the destination becomes visible to the vector processor until another scalar/vector memory synchronization instruction is performed.

When MSYNC faults, it is not ensured that all previously issued scalar and vector memory instructions have finished. In this case, the scalar processor writes no longword value to the scalar destination of the MFVP. Depending on the exception encountered by the vector processor, the MSYNC takes a vector processor disabled fault or memory management fault. Note that it is UNPREDICTABLE whether the vector processor is idle when the fault is generated. After the fault has been serviced, the MSYNC may be returned to through an REI.

\If the time necessary to execute a MSYNC is long, an implementation may allow the execution of the MSYNC to be interrupted so as to service an interrupt request. In this case, the MSYNC must be restartable after the interrupt is serviced. Allowing MSYNC to be interrupted is not a requirement; however, implementations should consider this option if the execution of MSYNC could significantly delay the servicing of device interrupts. At the moment, no current implementation is planning on doing this.\

Section 14.1.3.3 gives the necessary rules and examples to determine what vector control register elements and vector control register bits MSYNC depends upon.

14.3.2.2 Memory Activity Completion Synchronization (VMAC) -

Privileged software needs a way to ensure scalar/vector memory synchronization that will not result in any exceptions being reported. Reading the VMAC IPR with the privileged MFPR instruction is provided for these situations. It is especially useful for context switching.

Once a MFPR from VMAC is issued by the scalar processor, the scalar processor executes no further instructions until VMAC completes, which it does when:

1. all vector and scalar memory activities have ceased;

2. all resultant memory writes have been made visible to both the scalar and vector processor;

3. and a longword value (which is UNPREDICTABLE) is returned to the scalar processor.

After writing the longword value to the scalar destination of the MFPR, the scalar processor then proceeds to execute the next instruction. If the scalar destination is in memory, it is UNPREDICTABLE whether the new value of the destination becomes visible to the vector processor until another scalar/vector memory synchronization operation is performed.

As stated in section 14.3.2, Scalar/Vector Memory Synchronization, the ceasing of vector and scalar memory activities does not mean that previously issued vector memory instructions have completed. For example, consider a vector memory instruction which has suspended execution due to an asynchronous memory management exception or hardware error. Once it becomes suspended, the instruction will write no further elements and its memory activity will cease. As a result, a subsequently issued VMAC will complete as soon as those writes that were made by the memory instruction before it was suspended are visible to both the scalar and vector processor. But, after the completion of the VMAC, the memory instruction is not completed and remains suspended.

\If the time necessary to execute a VMAC is long, an implementation may allow the execution of the VMAC to be interrupted so as to service an interrupt request. In this case, the VMAC must be restartable after the interrupt is serviced. Allowing VMAC to be interrupted is not a requirement; however, implementations should consider this option if the execution of VMAC could significantly delay the servicing of device interrupts. At the moment, no current implementation is planning on doing this.\

Vector arithmetic and memory management exceptions of previous vector instructions never fault an MFPR-from-VMAC and never suspend its execution.

14.3.3 Other Synchronization Between the Scalar and Vector Processors

o In the absence of pending vector arithmetic exceptions, reading a vector control register using the MFVP instruction waits for all previous writes to that register to complete. In addition, the scalar processor must wait for the MFVP result to be written before processing other instructions. An MFVP instruction that reads a vector control register must fault if there is any unreported exception that has occurred in the production of the value of the control register.

o Writing to VTBIA or VSAR with MTPR causes the new state of the changed vector IPR to affect the execution of all subsequently issued vector instructions.

- Reading from VPSR with MFPR after writing to VPSR with MTPR causes the new state of VPSR (and VAER if cleared by VPSR<RST>) to affect the execution of subsequently issued vector instructions. \ This synchronization mechanism for VPSR was added to ensure that the new value of VPSR is received by all the appropriate logic on both the vector and scalar processor before another vector instruction is issued.\

14.3.4 Memory Synchronization Within the Vector Processor (VSYNC)

The vector processor may concurrently execute a number of vector memory instructions through the use of multiple load/store paths to memory. When it is necessary to synchronize the accesses of multiple vector memory instructions the MSYNC instruction can be used; however, there are cases for which this instruction does more than is needed. If it is known that only synchronization between the memory accesses of vector instructions is required, the VSYNC instruction is more efficient.

VSYNC orders the conflicting memory accesses of vector-memory instructions issued after VSYNC with those of vector-memory instructions issued before VSYNC. Specifically, VSYNC forces the access of a memory location by any subsequent vector-memory instruction to wait for (depend upon) the completion of all prior conflicting accesses of that location by previous vector-memory instructions.

VSYNC does not have any synchronizing effect between scalar and vector memory access instructions. VSYNC also has no synchronizing effect between vector load instructions because multiple load accesses cannot conflict. It also does not ensure that previous vector memory management exceptions are reported to the scalar processor. \ VSYNC can be a no-op instruction in an implementation that has a single vector load/store path to memory. \

14.3.5 Required Use of Memory Synchronization Instructions

Table 14-9 shows for all possible pairs of vector or scalar reads and writes to a common memory location, whether one of the scalar/vector memory synchronization instructions or the VSYNC instruction must be issued after the first reference and before the second. Since the MSYNC instruction also includes the VSYNC function, it can always be used instead of VSYNC.

In general, these rules apply to any sequence of instructions that access a common memory location, no matter how many other vector or scalar instructions are issued between the first instruction that accesses the common location and the second instruction that accesses the same location. For example, the code sequence shown below depicts a vector load followed by a scalar write to the same memory location. Between these two instructions are other scalar/vector instructions that do not access the common memory location. A scalar/vector memory synchronization instruction (MSYNC or VMAC) must be executed sometime after the vector read and before the scalar write to the common location. (Here MSYNC is shown.)

```
VLDL   A, #4, V0
          .
other scalar/vector instructions
that do not access A
          .
MSYNC  Dst
MOVL   R0, A
```

In most cases, MSYNC is the preferred method for ensuring scalar/vector memory synchronization. However, there are special cases, usually encountered by an operating system, when VMAC is more appropriate.

Cases when Scalar/vector memory synchronization is required:

- o After a vector instruction that stores to memory and before a peripheral (I/O) data transfer of the stored location is initiated by an application program. This ensures that the value stored will be transferred to the output device. The application must ensure that this requirement is met by using MSYNC. Using VMAC in this case is not sufficient because unlike MSYNC, VMAC does not ensure that all previous vector memory instructions have successfully completed.

- o After a vector instruction that stores to memory and before the associated scalar processor can execute a HALT. This ensures that a read or modify by another processor will access the updated memory value. VMAC is the preferred method for this case.

- o Before the vector processor state is saved as a result of power failure. A read or modify of the same memory 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). Also, software is responsible for saving any pending vector processor exception status. VMAC is the preferred method for this case.

- o Before a context switch. Software is responsible for ensuring that the vector processor has completed all its memory accesses before performing a context switch. Software is also responsible for saving any pending vector processor exception status. VMAC is the preferred method for this case.

The scalar/vector memory synchronization instructions are the only ones that guarantee that the memory operations of the vector and scalar processors are synchronized. Writes to I/O space, changes in access mode, machine checks, interprocessor interrupts, execution of a HALT, REI, or interlocked instruction do not make the results of vector instructions that write to memory visible to the scalar processor, I/O subsystem, or other processors. Execution of a scalar/vector memory synchronization instruction must precede any of these mechanisms to ensure synchronization of all system components.

Table 14-9: When Scalar/Vector Memory Synchronization (M) Or VSYNC (V) Is Required Between Instructions That Reference The Same Memory Location

First Reference	Scalar	Scalar	Vector	Vector
Second Reference	Scalar	Vector	Scalar	Vector
Operation Sequence				
Read Read	No (1,2)	No (1)	No (1)	No (1)
Read Write	No (2)	No (3)	M	V (5)
Write Read	No (2)	M (4)	M	V
Write Write	No (2)	M (4)	M	V

Key:
1 - Scalar/vector memory synchronization or VSYNC is never required between two read accesses to a memory location.
2 - Scalar/vector memory synchronization is never required between two accesses by the VAX scalar processor to a memory location.
3 - The scalar read is synchronous and will have completed before a vector memory operation is issued.
4 - Although a scalar write is a synchronous instruction, scalar/vector memory synchronization is required to ensure that the written data is visible to the vector processor before the vector memory reference is executed.
5 - See the section on Required Use of Memory Synchronization Instructions for the conditions when VSYNC is not required between a vector memory read/write pair.

14.3.5.1 When VSYNC Is Not Required -

There exist conditions when VSYNC is not required between conflicting vector memory accesses. A VSYNC is not required before a vector memory store instruction (VST/VSCAT) if, for each memory location to be accessed by the store, both of the following conditions are met:

1. Each of the store’s accesses to the location does not conflict with any access to the location by previously-issued vector store instructions. Conflict is avoided in this case because either:

   - the location is not shared; or

   - all accesses to the location by previous store instructions were forced to complete by the issue of an MSYNC or VMAC.

2. Each of the store’s accesses to the location does not conflict with any access to the location by previously-issued vector load (VLD/VGATH) instructions. Conflict is avoided in this case because:

   - the location is not shared; or

   - all accesses to the location by previous load instructions were forced to complete by the issue of an MSYNC or VMAC; or

   - each of the store’s accesses to the location depends on the completion (as seen by the vector processor) of all accesses to the location by previous LOAD instructions. (The examples immediately below demonstrate this concept.)

In all other cases of conflicting vector memory accesses, VSYNC is necessary to ensure correct results.

In the following examples, VSYNC is not required because both of the above conditions have been met for each location accessed by the store instruction:

```
1.  VLDL        A, #4, V0           2.  VLDL      A, #4, V0
    VSTL        V0, A, #4               VSSUBL    R0, V0, V1
                                        VSTL      V1, A, #4
```

```
3.  VLDL/0       A, #4 ,V0
    VSMULL/0     #3, V0, V0
    VLDL/1       A, #4 ,V1
    VVMULL/1     V1, V1, V1
    VVMERGE/1    V1, V0, V2
    VSTL         V2, A, #4
```

```
4.  VLDL         A, #4, V0
    VSGTRF       #0, V0
    VLDL/1       B, #4, V1
    VLDL/0       C, #4, V2
    VVMERGE/0    V2, V1, V3
    VSTL         V3, A, #4
```

In the following examples, VSYNC is required before the vector memory store instruction:

```
1.  VLDL/1       A,#4,V0
    VSLSSL       #0,V1
    VSYNC
    VSTL/1       V1,A,#4
```

If the VSYNC is not included, V0 could contain incorrect data at the end of the sequence since the vector processor is allowed to begin the VSTL before the VLDL is finished. This is because there is no dependence between the VMR value used by the VLDL and the VSTL.

```
2.  VLDL         A, #4, V0
    VVMERGE/0    V0, V1, V1
    VSYNC
    VSTL         V1, A, #4
```

Unless the programmer can ensure that the VMR mask being used by the VVMERGE will force the access of each location by the VSTL to depend on the access to that location by the VLDL, a VSYNC is required. Note that in general, when masked operates provide a conditional path of dependence between conflicting memory accesses, a VSYNC is usually necessary to ensure correct results.

```
3.  VSTL         V1, A, #4
    MTVLR        #32
    VSYNC
    VLDL         A+128, #4, V2
```

In this example, the VSTL writes locations A to A+255 and the VLDL reads locations A+128 to A+255. Without the VSYNC, the vector processor is allowed to start reading locations A+128 to A+255 for the VLDL before the vector processor completes (or even starts) writing locations A+128 to A+255 for the VSTL. Consequently, V2[0:31] will not contain V1[32:63], which is the intended result.

Note that the rules on when VSYNC is not required (found on page 14-38) only apply to waiving the use of VSYNC prior to VST/VSCAT instructions.

4.
```
VGATHL       A, V2, V0       ; let at least two elements
                             ; of V2 be equal
VVMULL       V9, V0, V1
VSYNC
VSCATL       V1, A, V2
```

The VSYNC is needed in this example because the VSCATL may store elements of V1 into a common location before the VGATHL has finished loading that location into all the appropriate elements of V0. As a result, elements of V0 fetched from the same location may be unequal. Suppose in the example that V2[0] = V2[63] = 0 and that the original value of location A before the sequence starts is X. Then it is possible without the VSYNC that V0[63] = X*V9[0] and that (A)= V1[63] = V9[63]*V9[0]*X after the sequence completes.

5.
```
VLDL         A, #0, V0
VVMULL       V9, V0, V1
VSYNC
VSTL         V1, A, #0
```

The VSYNC is needed in this example because the VSTL may store elements of V1 into A before the VLDL has finished loading all elements of V0 from A. As a result, the elements of V0 may be unequal and so produce incorrect results.

14.4 MEMORY MANAGEMENT

The vector processor may include its own translation buffer and maintain its own copies of SBR, SLR, P0BR, P0LR, P1BR, and P1LR as a group, or may use the scalar processor's memory management unit. Hardware implementations must ensure that MTPR to these registers update the copy retained by the vector processor. Changes to P0BR, P0LR, P1BR, and P1LR due to a LDPCTX do not update the copies in the vector processor. Before software enables the vector processor again, explicit MTPRs to P0BR, P0LR, P1BR, and P1LR are required to guarantee correct operation. \ This is necessary because the process page tables may have been swapped out and back into another part of system virtual address space since the vector processor was disabled. \

An MTPR to TBIS must also invalidate the corresponding TB entry in the vector processor, and an MTPR to TBIA must also invalidate the entire TB in the vector processor. However, the vector TB is not invalidated by a LDPCTX instruction. Software can use an MTPR to the Vector TB Invalidate All (VTBIA) register to invalidate only the vector TB. An MTPR to VTBIA results in no operation on a processor that uses a common TB for the scalar and vector processors.

\ This allows software to preserve the vector TB across context switches and yields better performance if the same process is rescheduled. \

Updates to memory management registers and invalidates of translation buffer entries in the vector processor take place even when the vector processor is disabled (VPSR<VEN> is clear). However, the vector processor may load translation buffer entries only when the vector processor is executing a vector memory access instruction.

\ Software must execute a scalar/vector memory synchronization instruction before PTE modification and the subsequent MTPR to TBIS, TBIA, or VTBIA. If MTPR to TBIS, TBIA, or VTBIA is issued prior to the scalar/vector memory synchronization instruction and PTE modification, the entries in the translation buffer may be invalidated before they are finished being used by the currently executing or pending vector instructions.

Note that the alternative of requiring the vector processor to perform scalar/vector memory synchronization as part of the MTPR to TBIS, TBIA, or VTBIA is not a sufficient solution. Since software would be clearing the PTE valid bit before the MTPR is executed, there is a chance that the vector processor could be setting the PTE modify bit at the same time, undoing the PTE change made by software. \

The vector processor implements the modify-fault option if its scalar processor implements the virtual-machine option. \For good performance, implementations which support VMS as the real machine operating system should set PTE<M> on both the scalar and vector processor.\

Vector memory access instructions must not be used to read or write page tables. If a vector instruction is used to read or write page tables, the results are UNPREDICTABLE.

Vector instructions are not allowed to reference I/O space. If a vector instruction references I/O space, the results are UNPREDICTABLE. \ The preferred, but not required, implementation is to generate an access control violation with the VIO bit of the memory management fault parameter set to distinguish the access control violation as an illegal vector I/O space reference. \

Issuing vector instructions with memory management disabled causes the operation of the vector processor to be UNDEFINED. Disabling memory management when the vector processor is busy (VPSR<BSY> is set) also causes the operation of the vector processor to be UNDEFINED.

14.5 HARDWARE ERRORS

A vector processor implementation may experience error conditions (such as chip malfunctions, parity errors, or bus errors) which prevent it from executing and completing instructions and from which it cannot recover through its own means. Such errors are termed hardware errors and may occur at anytime, even when the vector processor is already disabled. Vector processor hardware errors do not normally halt the scalar processor.

At some point after the error condition occurs, the vector processor reports the error to the scalar processor. The reporting may be accomplished through a machine check; or by disabling the vector processor, setting VPSR<IMP>, and generating a vector processor disabled fault when the next vector instruction is issued. After the error is reported, the appropriate software handler will be invoked to diagnose the vector processor and determine the severity of the hardware error and whether the vector processor can be restarted.

During execution, software may wish to force the reporting of hardware errors encountered by previous vector instructions before issuing further ones. This can be accomplished by reading the VMAC IPR and by waiting for VPSR<BSY> to become clear.

An MFPR from VMAC ensures that all pending vector memory instructions have finished or are suspended by an asynchronous memory management exception, and that all vector-processor hardware errors encountered by these instructions are reported by the time the MFPR completes.

- If the errors are reported by machine check, then the exception is taken either upon the VMAC itself, or upon the instruction immediately following the VMAC.

- If the errors are reported through VPSR<IMP>, the vector processor sets VPSR<IMP> and disables itself by the time the scalar processor completes VMAC. Subsequently, a vector processor disabled fault will occur when the next vector instruction is issued. A read of VPSR immediately after the VMAC completes will find the vector processor disabled and VPSR<IMP> set.

Waiting for VPSR<BSY> to become clear before issuing further instructions ensures that all previous non-memory access instructions have been finished or are suspended by an asynchronous memory management exception, and that all vector-processor hardware errors encountered by these instructions are reported by the time VPSR<BSY> becomes clear.

- If the errors are reported by machine check, then the exception is taken either upon the first instruction during which the new state of VPSR<BSY> becomes visible to the scalar processor or upon the instruction immediately thereafter.

- o If the errors are reported through VPSR<IMP>, the vector processor sets VPSR<IMP> and disables itself by the time it clears VPSR<BSY>. Subsequently, a vector processor disabled fault will occur when the next vector instruction is issued. The first MFPR instruction which reads VPSR<BSY> as clear will also read VPSR<VEN> as clear and VPSR<IMP> as set.

VMAC does not ensure that hardware errors encountered by pending non-memory-access instructions will be reported. Waiting for VPSR<BSY> to become clear does not ensure that vector-processor hardware errors encountered by vector memory instructions are reported.

Software can force the reporting of hardware errors encountered during the execution of previous vector instructions (both memory and non-memory) by waiting for VPSR<BSY> to become clear and then issuing an MFPR from VMAC. This technique can be used during scalar context switching to cause hardware errors resulting from the execution of vector instructions for the current process to be reported before that process is context-switched.

Change History:

Revision J. Rich Brunner, December 1989

- o Created chapter by taking sections on execution model and exception reporting from chapter 13.
- o Included chapter into SRM revision J.
- o Removed reference to SPTEP.
- o Described what was pushed on the stack by a vector processor disabled fault.
- o Expanded clarification text of Vector Arithmetic Exceptions.

APPENDIX A

OPCODE ASSIGNMENTS

Table A-1: Single Byte Opcodes

Hex	Mnemonic	Hex	Mnemonic
00	HALT	20	ADDP4
01	NOP	21	ADDP6
02	REI	22	SUBP4
03	BPT	23	SUBP6
04	RET	24	CVTPT
05	RSB	25	MULP
06	LDPCTX	26	CVTTP
07	SVPCTX	27	DIVP
08	CVTPS	28	MOVC3
09	CVTSP	29	CMPC3
0A	INDEX	2A	SCANC
0B	CRC	2B	SPANC
0C	PROBER	2C	MOVC5
0D	PROBEW	2D	CMPC5
0E	INSQUE	2E	MOVTC
0F	REMQUE	2F	MOVTUC
10	BSBB	30	BSBW
11	BRB	31	BRW
12	BNEQ, BNEQU	32	CVTWL
13	BEQL, BEQLU	33	CVTWB
14	BGTR	34	MOVP
15	BLEQ	35	CMPP3
16	JSB	36	CVTPL
17	JMP	37	CMPP4
18	BGEQ	38	EDITPC
19	BLSS	39	MATCHC
1A	BGTRU	3A	LOCC
1B	BLEQU	3B	SKPC
1C	BVC	3C	MOVZWL
1D	BVS	3D	ACBW
1E	BGEQU, BCC	3E	MOVAW
1F	BLSSU, BCS	3F	PUSHAW

Table A-1: Single Byte Opcodes (cont)

Hex	Mnemonic	Hex	Mnemonic
40	ADDF2	60	ADDD2
41	ADDF3	61	ADDD3
42	SUBF2	62	SUBD2
43	SUBF3	63	SUBD3
44	MULF2	64	MULD2
45	MULF3	65	MULD3
46	DIVF2	66	DIVD2
47	DIVF3	67	DIVD3
48	CVTFB	68	CVTDB
49	CVTFW	69	CVTDW
4A	CVTFL	6A	CVTDL
4B	CVTRFL	6B	CVTRDL
4C	CVTBF	6C	CVTBD
4D	CVTWF	6D	CVTWD
4E	CVTLF	6E	CVTLD
4F	ACBF	6F	ACBD
50	MOVF	70	MOVD
51	CMPF	71	CMPD
52	MNEGF	72	MNEGD
53	TSTF	73	TSTD
54	EMODF	74	EMODD
55	POLYF	75	POLYD
56	CVTFD	76	CVTDF
57	Reserved to DIGITAL	77	Reserved to DIGITAL
58	ADAWI	78	ASHL
59	Reserved to DIGITAL	79	ASHQ
5A	Reserved to DIGITAL	7A	EMUL
5B	Reserved to DIGITAL	7B	EDIV
5C	INSQHI	7C	CLRQ, CLRD, CLRG
5D	INSQTI	7D	MOVQ
5E	REMQHI	7E	MOVAQ, MOVAD, MOVAG
5F	REMQTI	7F	PUSHAQ, PUSHAD, PUSHAG

Table A-1: Single Byte Opcodes (cont)

Hex	Mnemonic	Hex	Mnemonic
80	ADDB2	A0	ADDW2
81	ADDB3	A1	ADDW3
82	SUBB2	A2	SUBW2
83	SUBB3	A3	SUBW3
84	MULB2	A4	MULW2
85	MULB3	A5	MULW3
86	DIVB2	A6	DIVW2
87	DIVB3	A7	DIVW3
88	BISB2	A8	BISW2
89	BISB3	A9	BISW3
8A	BICB2	AA	BICW2
8B	BICB3	AB	BICW3
8C	XORB2	AC	XORW2
8D	XORB3	AD	XORW3
8E	MNEGB	AE	MNEGW
8F	CASEB	AF	CASEW
90	MOVB	B0	MOVW
91	CMPB	B1	CMPW
92	MCOMB	B2	MCOMW
93	BITB	B3	BITW
94	CLRB	B4	CLRW
95	TSTB	B5	TSTW
96	INCB	B6	INCW
97	DECB	B7	DECW
98	CVTBL	B8	BISPSW
99	CVTBW	B9	BICPSW
9A	MOVZBL	BA	POPR
9B	MOVZBW	BB	PUSHR
9C	ROTL	BC	CHMK
9D	ACBB	BD	CHME
9E	MOVAB	BE	CHMS
9F	PUSHAB	BF	CHMU

Table A-1: Single Byte Opcodes (cont)

Hex	Mnemonic	Hex	Mnemonic
C0	ADDL2	E0	BBS
C1	ADDL3	E1	BBC
C2	SUBL2	E2	BBSS
C3	SUBL3	E3	BBCS
C4	MULL2	E4	BBSC
C5	MULL3	E5	BBCC
C6	DIVL2	E6	BBSSI
C7	DIVL3	E7	BBCCI
C8	BISL2	E8	BLBS
C9	BISL3	E9	BLBC
CA	BICL2	EA	FFS
CB	BICL3	EB	FFC
CC	XORL2	EC	CMPV
CD	XORL3	ED	CMPZV
CE	MNEGL	EE	EXTV
CF	CASEL	EF	EXTZV
D0	MOVL	F0	INSV
D1	CMPL	F1	ACBL
D2	MCOML	F2	AOBLSS
D3	BITL	F3	AOBLEQ
D4	CLRL, CLRF	F4	SOBGEQ
D5	TSTL	F5	SOBGTR
D6	INCL	F6	CVTLB
D7	DECL	F7	CVTLW
D8	ADWC	F8	ASHP
D9	SBWC	F9	CVTLP
DA	MTPR	FA	CALLG
DB	MFPR	FB	CALLS
DC	MOVPSL	FC	XFC
DD	PUSHL	FD	Two-Byte Opcode
DE	MOVAL, MOVAF	FE	Two-Byte Opcode
DF	PUSHAL, PUSHAF	FF	Two-Byte Opcode

Table A-2: Two-Byte Opcodes

Hex	Mnemonic	Hex	Mnemonic
00FD	Reserved to DIGITAL	50FD	MOVG
01FD	Reserved to DIGITAL	51FD	CMPG
02FD	WAIT	52FD	MNEGG
		53FD	TSTG
		54FD	EMODG
03FD		55FD	POLYG
to		56FD	CVTGH
2FFD	Reserved to DIGITAL	57FD	Reserved to DIGITAL
30FD	Reserved to DIGITAL	58FD	Reserved to DIGITAL
31FD	MFVP	59FD	Reserved to DIGITAL
32FD	CVTDH	5AFD	Reserved to DIGITAL
33FD	CVTGF	5BFD	Reserved to DIGITAL
34FD	VLDL	5CFD	Reserved to DIGITAL
35FD	VGATHL	5DFD	Reserved to DIGITAL
36FD	VLDQ	5EFD	Reserved to DIGITAL
37FD	VGATHQ	5FFD	Reserved to DIGITAL
38FD	Reserved to DIGITAL	60FD	ADDH2
39FD	Reserved to DIGITAL	61FD	ADDH3
3AFD	Reserved to DIGITAL	62FD	SUBH2
3BFD	Reserved to DIGITAL	63FD	SUBH3
3CFD	Reserved to DIGITAL	64FD	MULH2
3DFD	Reserved to DIGITAL	65FD	MULH3
3EFD	Reserved to DIGITAL	66FD	DIVH2
3FFD	Reserved to DIGITAL	67FD	DIVH3
40FD	ADDG2	68FD	CVTHB
41FD	ADDG3	69FD	CVTHW
42FD	SUBG2	6AFD	CVTHL
43FD	SUBG3	6BFD	CVTRHL
44FD	MULG2	6CFD	CVTBH
45FD	MULG3	6DFD	CVTWH
46FD	DIVG2	6EFD	CVTLH
47FD	DIVG3	6FFD	ACBH
48FD	CVTGB	70FD	MOVH
49FD	CVTGW	71FD	CMPH
4AFD	CVTGL	72FD	MNEGH
4BFD	CVTRGL	73FD	TSTH
4CFD	CVTBG	74FD	EMODH
4DFD	CVTWG	75FD	POLYH
4EFD	CVTLG	76FD	CVTHG
4FFD	ACBG	77FD	Reserved to DIGITAL

Table A-2: Two-Byte Opcodes (cont)

Hex	Mnemonic	Hex	Mnemonic
78FD	Reserved to DIGITAL	A0FD	VVMULL
79FD	Reserved to DIGITAL	A1FD	VSMULL
7AFD	Reserved to DIGITAL	A2FD	VVMULG
7BFD	Reserved to DIGITAL	A3FD	VSMULG
7CFD	CLRH, CLRO	A4FD	VVMULF
7DFD	MOVO	A5FD	VSMULF
7EFD	MOVAH, MOVAO	A6FD	VVMULD
7FFD	PUSHAH, PUSHAO	A7FD	VSMULD
80FD	VVADDL	A8FD	VSYNC
81FD	VSADDL	A9FD	MTVP
82FD	VVADDG	AAFD	VVDIVG
83FD	VSADDG	ABFD	VSDIVG
84FD	VVADDF	ACFD	VVDIVF
85FD	VSADDF	ADFD	VSDIVF
86FD	VVADDD	AEFD	VVDIVD
87FD	VSADDD	AFFD	VSDIVD
88FD	VVSUBL	B0FD	Reserved to DIGITAL
89FD	VSSUBL	B1FD	Reserved to DIGITAL
8AFD	VVSUBG	B2FD	Reserved to DIGITAL
8BFD	VSSUBG	B3FD	Reserved to DIGITAL
8CFD	VVSUBF	B4FD	Reserved to DIGITAL
8DFD	VSSUBF	B5FD	Reserved to DIGITAL
8EFD	VVSUBD	B6FD	Reserved to DIGITAL
8FFD	VSSUBD	B7FD	Reserved to DIGITAL
90FD	Reserved to DIGITAL	B8FD	Reserved to DIGITAL
91FD	Reserved to DIGITAL	B9FD	Reserved to DIGITAL
92FD	Reserved to DIGITAL	BAFD	Reserved to DIGITAL
93FD	Reserved to DIGITAL	BBFD	Reserved to DIGITAL
94FD	Reserved to DIGITAL	BCFD	Reserved to DIGITAL
95FD	Reserved to DIGITAL	BDFD	Reserved to DIGITAL
96FD	Reserved to DIGITAL	BEFD	Reserved to DIGITAL
97FD	Reserved to DIGITAL	BFFD	Reserved to DIGITAL
98FD	CVTFH	C0FD	VVCMPL
99FD	CVTFG	C1FD	VSCMPL
9AFD	PROBEVMR	C2FD	VVCMPG
9BFD	PROBEVMW	C3FD	VSCMPG
9CFD	VSTL	C4FD	VVCMPF
9DFD	VSCATL	C5FD	VSCMPF
9EFD	VSTQ	C6FD	VVCMPD
9FFD	VSCATQ	C7FD	VSCMPD

Table A-2: Two-Byte Opcodes (cont)

Hex	Mnemonic	Hex	Mnemonic
C8FD	VVBISL	F0FD	Reserved to DIGITAL
C9FD	VSBISL	F1FD	Reserved to DIGITAL
CAFD	Illegal Vector Opcode	F2FD	Reserved to DIGITAL
CBFD	Illegal Vector Opcode	F3FD	Reserved to DIGITAL
CCFD	VVBICL	F4FD	Reserved to DIGITAL
CDFD	VSBICL	F5FD	Reserved to DIGITAL
CEFD	Illegal Vector Opcode	F6FD	CVTHF
CFFD	Illegal Vector Opcode	F7FD	CVTHD
D0FD	Reserved to DIGITAL	F8FD	Reserved to DIGITAL
D1FD	Reserved to DIGITAL	F9FD	Reserved to DIGITAL
D2FD	Reserved to DIGITAL	FAFD	Reserved to DIGITAL
D3FD	Reserved to DIGITAL	FBFD	Reserved to DIGITAL
D4FD	Reserved to DIGITAL	FCFD	Reserved to DIGITAL
D5FD	Reserved to DIGITAL	FDFD	Reserved to DIGITAL
D6FD	Reserved to DIGITAL	FEFD	Reserved to DIGITAL
D7FD	Reserved to DIGITAL	FFFD	Reserved to DIGITAL
D8FD	Reserved to DIGITAL	00FE	
D9FD	Reserved to DIGITAL	to	
DAFD	Reserved to DIGITAL	FFFE	Reserved to DIGITAL
DBFD	Reserved to DIGITAL		
DCFD	Reserved to DIGITAL		
DDFD	Reserved to DIGITAL	00FF	
DEFD	Reserved to DIGITAL	to	
DFFD	Reserved to DIGITAL	F7FF	Reserved to DIGITAL
E0FD	VVSRLL	F8FF	Reserved to DIGITAL
E1FD	VSSRLL	F9FF	Reserved to DIGITAL
E2FD	Illegal Vector Opcode	FAFF	Reserved to DIGITAL
E3FD	Illegal Vector Opcode	FBFF	Reserved to DIGITAL
E4FD	VVSLLL	FCFF	Reserved to DIGITAL
E5FD	VSSLLL	FDFF	BUGL
E6FD	Illegal Vector Opcode	FEFF	BUGW
E7FD	Illegal Vector Opcode	FFFF	Reserved for all time
E8FD	VVXORL		
E9FD	VSXORL		
EAFD	Illegal Vector Opcode		
EBFD	Illegal Vector Opcode		
ECFD	VVCVT		
EDFD	IOTA		
EEFD	VVMERGE		
EFFD	VSMERGE		

Table A-3: Vector Opcodes

Hex	Mnemonic	Hex	Mnemonic
31FD	MFVP	ABFD	VSDIVG
34FD	VLDL	ACFD	VVDIVF
35FD	VGATHL	ADFD	VSDIVF
36FD	VLDQ	AEFD	VVDIVD
37FD	VGATHQ	AFFD	VSDIVD
80FD	VVADDL	C0FD	VVCMPL
81FD	VSADDL	C1FD	VSCMPL
82FD	VVADDG	C2FD	VVCMPG
83FD	VSADDG	C3FD	VSCMPG
84FD	VVADDF	C4FD	VVCMPF
85FD	VSADDF	C5FD	VSCMPF
86FD	VVADDD	C6FD	VVCMPD
87FD	VSADDD	C7FD	VSCMPD
88FD	VVSUBL	C8FD	VVBISL
89FD	VSSUBL	C9FD	VSBISL
8AFD	VVSUBG	CAFD	Illegal Vector Opcode
8BFD	VSSUBG	CBFD	Illegal Vector Opcode
8CFD	VVSUBF	CCFD	VVBICL
8DFD	VSSUBF	CDFD	VSBICL
8EFD	VVSUBD	CEFD	Illegal Vector Opcode
8FFD	VSSUBD	CFFD	Illegal Vector Opcode
9CFD	VSTL	E0FD	VVSRLL
9DFD	VSCATL	E1FD	VSSRLL
9EFD	VSTQ	E2FD	Illegal Vector Opcode
9FFD	VSCATQ	E3FD	Illegal Vector Opcode
		E4FD	VVSLLL
A0FD	VVMULL	E5FD	VSSLLL
A1FD	VSMULL	E6FD	Illegal Vector Opcode
A2FD	VVMULG	E7FD	Illegal Vector Opcode
A3FD	VSMULG	E8FD	VVXORL
A4FD	VVMULF	E9FD	VSXORL
A5FD	VSMULF	EAFD	Illegal Vector Opcode
A6FD	VVMULD	EBFD	Illegal Vector Opcode
A7FD	VSMULD	ECFD	VVCVT
A8FD	VSYNC	EDFD	IOTA
A9FD	MTVP	EEFD	VVMERGE
AAFD	VVDIVG	EFFD	VSMERGE

\Because CVAX and Rigel multiplex the IPLA based on the low-order bits of the opcode (2 bits for CVAX and 3 bits for Rigel), related classes of instructions will flow down COLUMNS of the opcode chart, rather than across in rows. Thus for the vector operate instructions:

```
byte2<1:0>   =  00 --> longword vector op vector
                01 --> longword scalar op vector
                10 --> quadword vector op vector
                11 --> quadword scalar op vector

byte2<6:5,3> =  000 --> add
                001 --> subtract
                010 --> multiply
                011 --> divide
                100 --> compare
                101 --> bic/bis
                110 --> shift
                111 --> misc
```

Table A-4: Vector Opcode Groupings By Format

opcode	cntrl.rw, base.ab, stride.rl
VLDL	34FD
VLDQ	36FD
VSTL	9CFD
VSTQ	9EFD

opcode	cntrl.rw, base.ab
VGATHL	35FD
VGATHQ	37FD
VSCATL	9DFD
VSCATQ	9FFD

opcode	cntrl.rw or regnum.rw, scal.rl
VSADDL	81FD
VSCMPL	C1FD
VSMULL	A1FD
VSSUBL	89FD
VSBICL	CDFD
VSBISL	C9FD
VSXORL	E9FD
VSSLLL	E5FD
VSSRLL	E1FD
VSADDF	85FD
VSSUBF	8DFD
VSDIVF	ADFD
VSMULF	A5FD
VSCMPF	C5FD
IOTA	EDFD
MTVP	A9FD

opcode	cntrl.rw, scal.rq
VSADDD	87FD
VSADDG	83FD
VSCMPD	C7FD
VSCMPG	C3FD
VSDIVD	AFFD
VSDIVG	ABFD
VSMULD	A7FD
VSMULG	A3FD
VSSUBD	8FFD
VSSUBG	8BFD
VSMERGE	EFFD

opcode	cntrl.rw or regnum.rw
VVADDL	80FD
VVCMPL	C0FD
VVMULL	A0FD
VVSUBL	88FD
VVBICL	CCFD
VVBISL	C8FD
VVSLLL	E4FD
VVSRLL	E0FD
VVXORL	E8FD
VVADDF	84FD
VVADDD	86FD
VVADDG	82FD
VVCMPF	C4FD
VVCMPD	C6FD
VVCMPG	C2FD
VVDIVF	ACFD
VVDIVD	AEFD
VVDIVG	AAFD
VVMULF	A4FD
VVMULD	A6FD
VVMULG	A2FD
VVSUBF	8CFD
VVSUBD	8EFD
VVSUBG	8AFD
VVMERGE	EEFD
VVCVT	ECFD
VSYNC	A8FD

opcode	regnum.rw, dst.wl
MFVP	31FD

Change History:

Revision J. Rich Brunner, December 1989.

- Added in Vector Opcodes and Vector Opcode tables.
- Restructured opcode tables

Revision H. Tim Leonard, May 1987.

- Remove binary.
- Add opcodes for VM support.

Revision E. Tim Leonard, September, 1986.

- Clarify the listing of the opcodes that are "Reserved to DIGITAL."

Revision D. Tim Leonard, February 1985.

- Change the revision number to correspond to DEC Standard 032 rev number.
- Rename to Appendix A.
- Move the section on notation to the beginning of Chapter 4.
- Move the section on Instructions Usable to Reference I/O Space to Chapter 8.

Revision 18, no I/O in compatibility mode. Tom Eggers, 26 July 1982.
Revision 17, I/O instructions and I/O access. Tom Eggers, 17 June 1980.
Revision 16. Dileep Bhandarkar, 25 October 1978.

- Add 2-byte opcodes for G_floating and H_floating.
- Fix CRC destination.
- Fix POLYD implied operands.
- Add interlocked queue instructions.

Revision 15. Bill Strecker, 9 March 1977.

- Change name of EDITN to EDITPC (EDITPC ECO).
- Delete option and destination length operands of EDIT (EDITPC ECO).
- Change from zoned to packed instructions (decimal data ECO).
- Add CLRF and CLRD as same as CLRL and CLRQ.
- Add Rounding to ASHP.
- Correct order of field reference on EXTV.
- Change operands of POLY Instruction
- Add CVT{SP,PS}; change xxN to xxT; add INDEX.
- Add ADAWI.

Revision 14, results of April 1 through 9 meeting. Bill Strecker, 3 June 1976.

- Add MOVQ.
- Remove CLRF.
- Add ADDA2, ADDA3, SUBA2, SUBA3.
- Remove CHOPF, CHOPD.
- Correct EMOD operands per ECO 18.
- Remove POLYF, POLYD.
- Change MOVP, PUSHP to MOVA, PUSHA.
- Remove CMPA, DIFA, ADTA, SBFA.
- Correct conditional branches per ECO 17.

- o Add BBSSI, BBCCI.
- o Remove MSx, MSPx.
- o Add XFC.
- o Remove ECMA.
- o Add INSQUE, REMQUE.
- o Add MOVTUC, MATCHC.
- o Add CRC per ECO 12.
- o Correct CRC operands.
- o Remove MOVU, CVTLU, CVTPU, ASHU; change names to MOVN, CVTLN, CVTPN, ASHN.
- o Remove MULN4, DIVN4; change names to MULN, DIVN.
- o Change name to EDITN.
- o Add CHMK, CHME, CHMS, CHMU, CHMI.
- o Add PRBPR, PRBPRW.
- o Add LDPC, SVPC.
- o Add MTPR, MFPR.
- o Replace opcode encoding to minimize decode logic (April 1976).
- o Change names to LDPCTX, SVPCTX.
- o Change MTPR and MFPR from .wl to .rl.
- o Change queue context from .aq to .ab.
- o Correct FFS and FFC to .wl.
- o Correct EMOD to mulrx.
- o Correct typo on CLR.
- o Add section on instruction interruptibility.
- o Change operand names to agree with chapters 4 through 9.
- o Add mode to PROBER and PROBEW and change names.
- o Remove CHMI.
- o Change MOVPSW to MOVPSL.
- o Remove ADDA, SUBA.
- o HALT and BPT are faults.
- o Add POLYF and POLYD.
- o Add SKPC.
- o Change EMODD to int.wl.
- o Add implicit operands.
- o Change field operand type from .ab to .vb.
- o Change name to BLBS and BLBC.
- o BPT is legal in all modes and on interrupt stack.
- o SVPCTX is legal in kernel mode.
- o Add final encoding (1-Jun-76).
- o Add CRC destination.
- o Expand I/O restriction rules.
- o Allow boot in I/O space.
- o Put {} around implied operands.
- o Add field implied operand on field instructions.
- o Add register implied operands on string and POLY.

Revision 13. Bill Strecker, 16 March 1976.

- o Change operands to EXTV, EXTZV, INSV, CMPV, CMPZV, FFC, FFS, BBx, BBxx, BLS, BLC per ECO 10.

Revision 12. Roger Gourd, 25 February 1976.

- o Change operand of CALLS per ECO 11.

Revision 11. Bill Strecker, 27 February 1976.

Revision 10. Bill Strecker, 26 February 1976.
Revision 9, conditional branch. Bill Strecker, 13 February 1976.

- o Add BME, BPN.
- o Remove BNEVER.
- o Add BSBB; change name to BSBW.
- o Change name to BRB, BRW.
- o Change opcode assignments of above and BLS, BLC, JMP, JSB.

Revision 7, opcode assignments added. Dave Rodgers, 22 January 1976.
Revision 6, memory management. Tom Hastings.

- o Add opcode assignments; add CRLF.

Revision 5, name changes. Bill Strecker, 2 February 1976.

- o Change name to EMUL, EDIV.
- o Change name to ASHQ.
- o Change name to MOVP, PUSHP, CMPA, ADTA, SBFA, DIFA.
- o Change name to FFC, FFS.
- o Change name to BR1, BR2.
- o Change name to BLS, BLC.
- o Change name to CALLG, RET.
- o Change name to MOVPSW.
- o Change name to EMOD.
- o Change name to MOVC, MOVTC, CMPC, SCANC, SPAN, LOCC.
- o Change name to CMPN, ADDN, SUBN, MULN, DIVN, CVTNL, CVTLS, CVTPS, ASHS.
- o Change name to MOVC3, MOVC5, etc.
- o Change name to EDITC.
- o Change name to MS, MPS.

Revision 4, changes to field, loop, and string. Dave Rodgers, 28 February 1977.
Revision 3, chapter 10 ECO #3. Steve Rothman, 9 January 1976.
Revision 2, results of pruning. Tom Hastings, 7 January 1976.

- o Change operands of INSV, EXTV, EXTZV, CMPV, CMPZV, FFZ, FFO, BBx, BBxx, BT, BF.
- o Remove AVP.
- o Change order of string descriptor.
- o Change order of operands in ACB, AOBLT, SOBGT.

Revision 1, review of Chapters 1 through 4. Tom Hastings, 19 December 1975.

APPENDIX B

IMPLEMENTATION DEPENDENCIES

The VAX family of processors shares a common architecture, including data types, instructions, addressing modes, and registers. Software written to depend only on these features will run on any VAX processor. Some software, however (typically operating system software), by necessity depends on features that vary from implementation to implementation.

This appendix describes individual VAX processors, in particular those features that are typically of interest to operating systems programmers. Such features include:

- Instruction set
- Layout of physical memory
- System control block
- Codes for the halt conditions
- Internal processor registers
- Contents of the machine-check stack frame
- Operations that are specified UNDEFINED or UNPREDICTABLE.

The information in this appendix is included as a convenience; limitations of time and space prevent the inclusion of complete descriptions of each implementation. For more complete and current information, refer to the engineering specifications of particular processors.

\Implementors should consult with the VAX Architecture Group when designing the implementation-dependent parts of a processor.\

B.1 IMPLEMENTATION OPTIONS

Some instructions, data types, and processor registers described in this manual may be omitted from VAX processors. For more detail, refer to Chapter 11, Implementation Options.

B.1.1 The Physical Address Space

VAX virtual addresses are 32 bits in length. Virtual addresses are translated into physical addresses by a function that is implementation dependent, and that depends on whether memory mapping is enabled, as described in Chapter 4, Memory Management.

VAX physical addresses are at most 34 bits in length, but processors may implement fewer address bits. The physical address space consists of two parts: memory space and I/O space. Memory space starts at address zero and continues to an implementation-dependent limit. I/O space begins at that limit and continues to the end of the physical address space. Neither memory space nor I/O space are necessarily filled and typically will be sparsely filled.

Both memory space and I/O space are addressed by bytes. Aligned and unaligned references to memory of byte, word, and longword size are supported. Only aligned longword references are necessarily supported to I/O space. References of other sizes may be supported on some implementations.

Typically, I/O space consists of several "adapter spaces" and one or more address spaces. The adapter spaces are sections of the address space set aside for the registers of various bus adapters and memory controllers. Many adapter spaces begin with an "adapter configuration register" which contains an adapter type code. This is for use by the operating system during power-up initialization to help it determine the system hardware configuration.

On systems with a UNIBUS interconnect, UNIBUS address spaces are sections of the I/O address space which directly map to a UNIBUS address space. UNIBUS addresses are 18 bits in length, so a UNIBUS address space is 256 kilobytes in length. Within the UNIBUS address space, the low 248 Kbytes is UNIBUS memory space. Typically, UNIBUS references to UNIBUS memory space are translated by a set of UNIBUS map registers to references in the VAX physical address space. This allows UNIBUS devices to directly access VAX physical memory.

B.1.2 The System Control Block

The system control block is a block of physical memory that contains vectors for exceptions and interrupts. Chapter 5 describes its format and interpretation. VAX processors may include exception and interrupt vectors in addition to those described in Chapter 5.

B.1.3 Halt Codes

Chapter 10 describes halting. When a VAX processor halts, the reason for the halt is saved in a halt code. A processor may report halt codes in addition to those described in Chapter 10.

B.1.4 Internal Processor Registers

Chapter 8, Privileged Registers, describes the internal processor register (IPR) address space and the registers found there on every machine. Processors may include IPRs in addition to those described in Chapter 8.

B.1.5 Machine Checks

Chapter 5, Interrupts and Exceptions, describes the overall format of the machine-check stack frame. Included in the stack frame is space for implementation-dependent error report information. The circumstances that cause machine-check are different for each processor, and the information reported is different as well.

B.1.6 UNPREDICTABLE and UNDEFINED

As used in this book, the terms UNPREDICTABLE and UNDEFINED have particular meanings. Results specified as UNPREDICTABLE may vary from one execution to the next. Software must not depend on any UNPREDICTABLE results. The results of an instruction include:

- Explicit destination operands (those with operand specifiers)
- Implicit destination operands
- Registers modified by operand specifier evaluation, including specifiers for implied operands
- PSL condition codes
- PSL<FPD>
- PSL<TP>, if PSL<T> was set at the beginning of the instruction
- PTE<M> for pages mapping write or modify type operands (PTE<M> will be set if the instruction modified the page, or if PTE<M> was set before the instruction started.)

PC and unlisted fields of the PSL are specifically excluded from this list. They are UNPREDICTABLE only when they appear as explicit or implicit operands.

UNPREDICTABLE results are constrained by memory mapping and access protection. That is, if correctly operating instructions cannot affect a memory location or privileged register, then an instruction with UNPREDICTABLE results cannot either.

A complete list of all UNPREDICTABLE results in the VAX Architecture exists but is too large to include within this specification. This list can easily be obtained by sending mail to the VAX Architecture group at EAGLE1::SRM.

UNDEFINED operations result from privileged software performing proscribed actions. The effects may be widespread and are not necessarily constrained by memory mapping or access control. UNDEFINED operations may affect the contents of memory, the operation of peripherals, and the operation of the processor. UNDEFINED operations are constrained only to not hang the processor and console. Control of the processor can be regained by reinitializing the processor from the console.

The complete list of UNDEFINED operations is implementation-dependent but includes the following operations below.

- Console START or CONTINUE after an error halt and before a processor initialization
- LDPCTX when the new kernel stack is invalid or inaccessible.
- When the operation of the VAX scalar processor becomes UNDEFINED, so does the operation of its associated vector processor. The converse is not true; when the operation of the vector processor becomes UNDEFINED, the operation of the scalar processor need not become UNDEFINED.
- If the console microprocessor cannot complete its own power-up initialization, the state of the console and that of the processor is UNDEFINED.
- If the SCBB points to I/O space or nonexistent memory when an exception or interrupt occurs, the operation of the processor is UNDEFINED. If SCBB contains a virtual address, the page or pages containing the SCB must be valid and accessible to kernel mode, or the operation of the processor is UNDEFINED.
- If part or all of the system page table resides in I/O space, in nonexistent memory, or beyond the end of the physical-address space while memory mapping is enabled, the operation of the processor is UNDEFINED.
- If part or all of either process page table is mapped into I/O space or nonexistent memory while memory mapping is enabled, the operation of the processor is UNDEFINED.
- References to PCB entries located in I/O space result in UNDEFINED behavior.
- References in the I/O space other than in UNIBUS spaces are UNDEFINED with respect to interlocking. This includes the BBSSI and BBCCI instructions.
- String, quadword, octaword, F_floating, D_floating, G_floating, H_floating, and field references in the I/O space result in UNDEFINED behavior.
- References to addresses beyond the end of the physical address space result in UNDEFINED behavior.
- Probing nonexistent memory (or addresses beyond the end of the physical address space) results in UNDEFINED processor operation.
- If the instruction stream comes within 512 bytes of nonexistent memory or the end of the physical address space when memory management is disabled, prefetching references by the processor may cause UNDEFINED behavior.
- Before enabling memory management, software must flush the entire translation buffer. If software does not follow this rule, the operation of the processor becomes UNDEFINED when memory management is enabled.
- Issuing vector instructions with memory management disabled causes the operation of the vector processor to be UNDEFINED. Disabling memory management when the vector processor is busy (VPSR<BSY> is set) also causes the operation of the vector

processor to be UNDEFINED.

- o SLR, P0LR, and P1LR values not in the range of 0 to 200000 (hex) inclusive are reserved values and result in UNDEFINED
- o The operation of the processor is UNDEFINED if process space page tables are read-only or no-access.
- o In kernel mode, the operation of the processor is UNDEFINED after the execution of an MTPR to P0BR or P1BR which writes a non-zero value to P0BR<8:0>.
- o If the processor holds a virtual address in PCBB, the PCB must not cross a page boundary, and the page containing the PCB must be valid, and the page must allow kernel write access, and (on processors that implement modify fault) the page must be marked "modified", or the operation of the processor is UNDEFINED.
- o After software changes a valid system PTE that maps any part of a process page table, software must flush the translation for the system page and then flush the translations of all valid process pages so mapped. If software writes such a process page before flushing its translation, the operation of the processor is UNDEFINED.
- o If IPL is lowered to zero when the processor is running on the interrupt stack, the operation of the processor is UNDEFINED.
- o For kernel-stack-not-valid abort and machine-check exception, if the SCB exception vector <1:0> is not 1, the operation of the processor is UNDEFINED.
- o Bits <1:0> in the CHMx and emulation exception vectors must be zero or the operation of the processor is UNDEFINED.
- o If writable control store does not exist or is not loaded, when an exception is reported through an SCB vector with bits <1:0> equal to 2, then the operation of the processor is UNDEFINED.
- o The operation of the processor becomes UNDEFINED when servicing an exception or interrupt whose SCB vector has bits <1:0> both set.
- o In kernel mode, the operation of the processor is UNDEFINED after execution of MTPR to a read-only register, MTPR to a nonexistent register, MTPR of a non-zero value to an MBZ field, or MTPR of a reserved value to a register.
- o In kernel mode, the operation of the processor is UNDEFINED after execution of MFPR from a register that does not exist, or after execution of MFPR from a write-only register.
- o The TBIS, TBIA, TBCHK, TBIASN, TBISYS processor registers are write only. The operation of MFPR from any of these registers is UNDEFINED.
- o

  Execution of MTPR src, _ PR$__ASTLVL with src<31:0> GEQU 5 results in UNDEFINED behavior.
- o If VPSR<RST> is set by software while VPSR<BSY> is set, the operation of the vector processor is UNDEFINED.

- When the processor is executing a virtual machine, the value of PSL is constrained by the value of VMPSL. Specifically,

  PSL<CUR_MOD> is equal to MAXU(VMPSL<CUR_MOD>, 1);
  PSL<PRV_MOD> is equal to MAXU(VMPSL<PRV_MOD>, 1);
  PSL<IPL> is zero; and
  PSL<IS> is zero.

  If any of these constraints are not met on entry into a virtual machine (that is, when REI loads a PSL with VM set), operation of the processor is UNDEFINED.

B.2 MICROVAX I

The MicroVAX I computer system is the first subset VAX. Announced in 1984, it is packaged in a box about 6 inches by 28 inches by 22 inches.

The MicroVAX I preceded the current instruction implementation rules. It comes in two versions; one includes F_floating and G_floating instructions, the other includes F_floating and D_floating instructions. Neither version includes CMPC5, which is now required. Both versions include the ACBx, EMODx, and POLYx instructions for the floating point data types supported. No other optional instructions or features are included.

Implementation-dependent features of the MicroVAX I are described in Figures B-2 through B-4 and Tables B-1 through B-4.

B.3 MICROVAX II

The MicroVAX II computer system is the first VAX with the processor on a single chip. F_floating, D_floating, and G_floating instructions are provided by a floating-point unit (another chip). The MicroVAX II preceded the current instruction implementation rules. It does not include CMPC3, CMPC5, LOCC, SKPC, SPANC, and SCANC, which are now required. It includes the ACBx, EMODx, and POLYx instructions for the three floating point data types supported. No other optional instructions or features are included.

Implementation-dependent features of the MicroVAX II are described in Figures B-5 through B-7 and Tables B-5 through B-8.

B.4 VAX-11/725

The VAX-11/725 computer system, announced in 1984, is a repackaged version of the VAX-11/730 processor. The cabinet is 25 inches high and 18 inches wide, and includes memory, two TU58 tape cartridge drives, and an RC25 disk.

B.5 VAX-11/730

The VAX-11/730 computer system, announced in 1982, was the third processor in the VAX family, and the first to include G_floating and H_floating as standard. It is packaged with two disks in a cabinet 42 inches tall and 22 inches wide.

The VAX-11/730 includes all the instructions, all the architecturally defined processor registers, and compatibility mode.

Implementation-dependent features of the VAX-11/730 are described in Figures B-8 through B-10 and Tables B-9 through B-12.

B.6 VAX-11/750

The VAX-11/750, announced in 1980, was the second processor in the VAX family. It is packaged in a cabinet 42 inches tall and 29 inches wide.

The VAX-11/750 includes all the instructions (G_floating and H_floating are available as an option), all architecturally defined processor registers, and compatibility mode.

Implementation-dependent features of the VAX-11/750 are described in Figures B-11 through B-13 and Tables B-13 through B-16.

B.7 VAX-11/780

The VAX-11/780 computer, announced in 1978, was the first processor of the VAX family. It is packaged in a cabinet 60 inches tall and 47 inches wide.

The VAX-11/780 includes all the instructions (G_floating and H_floating instructions are available as an option), all the architecturally defined processor registers, and compatibility mode.

The floating point architecture was revised in January 1979 to add two new data types (G_floating and H_floating) and to take faults instead of traps on floating exceptions for the original data types (F_floating and D_floating) as well as the two new data types. All VAX-11/780 processors have been modified to take faults instead of traps.

Implementation-dependent features of the VAX-11/780 are described in Figures B-14 through B-16 and Tables B-17 through B-20.

B.8 VAX-11/782

The VAX-11/782 computer system, announced in 1982, is a dual processor VAX-11/780 with shared memory. The cabinets containing the processor, I/O adapters, and shared memory are 60 inches tall and 190 inches wide.

B.9 VAX-11/785

The VAX-11/785 computer system, announced in 1984, is available as a field upgrade of the VAX 11/780. It is packaged in a cabinet 60 inches tall and 80 inches wide, including processor, memory, and I/O adapters. The VAX-11/785 is identical to the VAX-11/780 from the point of view of software, except that the VAX-11/785 has increased performance and has a bit set in the SID internal processor register, by which software can differentiate between the two processor types.

B.10 VAX 8200

The VAX 8200 computer system was announced in 1986, and it is packaged with two disks in a cabinet 42 inches tall and 22 inches wide.

The VAX 8200 includes all the instructions and architecturally defined processor registers, but does not include compatibility mode.

Implementation-dependent features of the VAX 8200 are described in Figures B-17 through B-19 and Tables B-21 through B-22.

B.11 VAX 8300

The VAX 8300 was announced in 1986, and is a dual-processor version of the VAX 8200, packaged in the same cabinet.

B.12 VAX 8500

The VAX 8500 was announced in 1986, and is a lower-performance, lower-cost version of the VAX 8700. It is packaged in a cabinet 60 inches tall and about 27 inches wide.

B.13 VAX 8600

The VAX 8600 was announced in 1984, and is packaged in a cabinet 60 inches tall and about 80 inches wide.

The VAX 8600 includes all the instructions, architecturally defined processor registers, and compatibility mode.

Implementation-dependent features of the VAX 8600 are described in Figures B-20 through B-22 and Tables B-23 through B-26.

B.14 VAX 8650

The VAX 8650 was announced in 1985, and is available as a field upgrade of the VAX 8600. The VAX 8650 is packaged in the same cabinet as the VAX 8600, and offers higher performance.

B.15 VAX 8700

The VAX 8700 is a single-processor version of the VAX 8800. It can be field-upgraded to a VAX 8800 by addition of a section processor.

B.16 VAX 8800

The dual-processor VAX 8800 was announced in 1986, and is the highest performance member of the VAX family. It is packaged in a cabinet 60 inches tall and about 80 inches wide.

The VAX 8800 includes all the instructions and architecturally defined processor registers, but does not include PDP-11 compatibility mode.

Implementation-dependent features of the VAX 8800 are described in Figures B-23 through B-25 and Tables B-27 through B-30.

B.17 VVAX

The security-kernel virtual VAX is an implementation of virtual machines on the VAX. It is also called the VVAX processor.

Because the VVAX is a virtual machine, one cannot point to the physical components that compose it. A virtual VAX does have a configuration, however, and does have an implementation of the implementation-dependent parts of the VAX architecture. Figure B-1 shows a conceptual block diagram of the configuration of a VVAX system.

The memory bus on a VVAX system is called the VVMI (VVAX Memory Interconnect). It connects the VVAX processor to memory and to the I/O bus. The I/O bus is called the VVII (VVAX I/O Interconnect). Data flow on the VVII is managed by a VVIA (VVAX I/O Interconnect Adapter). Connected to the VVII are the controllers for all I/O devices on the system.

```
+----------------+
|                |
| VVAX Processor |
|                |
+-------+--------+
        |
        |     +-------------------+
        |     |                   |
        +-----+ Read/write memory |
        |     |                   |
      V |     +-------------------+
      V |
      M |
      I |     +----------------+
        |     |    VVAX I/O    |                     VVII
        +-----+  Interconnect  +--+------- . . . ---------+--o
        |     | Adapter (VVIA) |  |                         |
        |     +----------------+  |                         |
        o                         |                         |
                           +------+------+           +------+------+
                           |    disk     |           |    other    |
                           | controller  | . . .     | controllers |
                           +-------------+           +-------------+
                              / ... \                   / ... \
```

Figure B-1 Conceptual Configuration of a VVAX System

B.17.1 VVAX Physical Address Space

As on a physical VAX, a physical address on the VVAX is 30 bits long, resulting in an address space size of 1024 MB. If a virtual machine makes a memory reference to a physical address with bits <31:30> not equal to zero, the operation of the VVAX processor is UNDEFINED. This is possible only when memory mapping is disabled in the virtual machine.

The physical address space is divided into three regions:

- read/write memory
- I/O space
- nonexistent memory

Read/write memory belongs to a single VVAX processor, which can both read from and write into that memory. No other VVAX can refer to that memory in any way.

I/O space is also private to a single VVAX, and the processor can both read from I/O space and write to it.

Nonexistent memory is memory address space beyond installed memory, and is not accessible to the processor. An attempt to access nonexistent memory generates a nonexistent memory machine check exception.

Read/write memory starts at physical address zero and has no holes. Read/write memory is immediately followed by I/O space, which is immediately followed by nonexistent memory.

To determine the size of its physical memory, a virtual machine reads the MEMSIZE internal processor register. Since I/O space immediately follows physical memory, MEMSIZE also gives the starting address of I/O space.

Figure B-26 is a picture of the physical address space on a VVAX processor.

B.17.2 VVAX Virtual Address Space

In most respects, the virtual address space of the VVAX processor conforms to DEC STD 032. However, three differences should be discussed:

1. Pages marked as readable only in kernel mode are also readable in executive mode, and pages marked as writable only in kernel mode are also writable in executive mode. In effect, kernel mode and executive mode have been combined into a single access mode for page protection.

2. The size of System space cannot be greater than the value in the Max_system_space_per_VM VAX/Secure Virtual System SYSGEN parameter. Any attempt to set the system length register (SLR) of the VVAX beyond Max_system_space_per_VM while memory management is enabled results in the virtual machine halting. This can occur on an MTPR to either SLR or MAPEN.

3. Total size of P0 space plus P1 space cannot be greater than the value in the Max_process_space_per_VM VAX/Secure Virtual System SYSGEN parameter. Any attempt to set the size of process space beyond this limit while memory management is enabled results in the virtual machine halting. This can occur on an MTPR instruction, on an LDPCTX instruction, or when enabling memory management.

B.17.3 VVAX MEMSIZE Register

This register indicates the amount of physical memory (in bytes) on this VVAX processor. The value is an integer multiple of 512. Writes to this register are ignored.

B.17.4 VVAX KCALL Register

This write-only register is used to communicate with the RMOS that underlies the virtual machine implementation. Reading from KCALL results in a reserved-operand fault. Writing into KCALL causes control to pass to the RMOS in order to perform some service on behalf of the virtual machine. Use of this register is analogous to use of the CHMx instruction in VAX/VMS.

B.17.5 VVAX ICCS Internal Processor Register

The VVAX supports a subset implementation of ICCS. When interval timer interrupts are enabled, an interval timer interrupt request will occur from time to time in each virtual machine, but the frequency of the interrupts is not predictable. The interrupt requests may occur more often or less often than every ten milliseconds of real time, although they are likely to occur less often.

To allow a virtual machine to have a fairly accurate estimate of the real time, the VAX/Secure Virtual System allows the VM to declare a software timer location quadword to the kernel. On every real interval timer interrupt, the kernel will write the current real time into the software timer locations of all VMs.

B.17.6 VVAX Instructions

The instruction groups supported by the VVAX processor depend, in part, on the instruction groups provided by the underlying real machine. Chapter 11 describes the implementation options. The VVAX supports those options if and only if they are implemented on the real machine, and exactly to the degree implemented by the real machine.

The following features are provided on the VVAX processor:

- o Console registers: RXCS, RXDB, TXCS, TXDB.

  Subset interval timer, that is, there are no NICR and ICR registers, and only the IE bit of the ICCS register is implemented.

- o No TODR register.

- o No support for self-virtualization, that is, the VVAX processor will not implement the virtualization group.

The processor type of the underlying real machine can be determined by issuing a kernel call.

Implementation-dependent features of the VVAX are described in Figures B-26 through B-30 and Tables B-31 through B-35.

```
                 +--------------------------------+
0000 0000:       |                                |
                 |        installed memory        |
                 |                                |
    :            | - - - - - - - - - - - - - - -  |
                 |                                |
                 |     memory address space       |
                 |   beyond installed memory      |
003F FFFF:       |                                |
                 +--------------------------------+
0040 0000:       |                                |
                 |            reserved            |
1FFF FFFF:       |                                |
                 +--------------------------------+
2000 0000:       |                                |
                 |        Q22-bus I/O space       |
2000 1FFF:       |                                |
                 +--------------------------------+
2000 2000:       |                                |
                 |            reserved            |
3FFF FFFF:       |                                |
                 +--------------------------------+
```

Figure B-2 MicroVAX I Physical Address Space

Table B-1: MicroVAX I Implementation-Dependent SCB Vectors

Offset	Vector Name	IPL	Notes
60	Write-bus timeout	1D	
C0	Interval timer	16	
200 - 3FC	Q22-bus interrupts	14-17	IPL corresponds to bus request levels 4 through 7.

Table B-2: MicroVAX I Halt Codes

Code	Meaning
1	Microverify succeeded
2	Processor halted by HALT button or console break
3	Power up
4	Interrupt stack not valid
5	Double machine check
6	HALT instruction executed
A	Change-mode from the interrupt stack
C	SCB-vector read error
FF	Microverify failed

Table B-3: MicroVAX I Implementation-Dependent IPRs

IPR	Mnemonic	Name
18	ICCS	Interval-clock control and status (1)
19	NICR	Next interval count (2)
1A	ICR	Interval count (2)
1B	TODR	Time-of-year clock (2)
24	TBDR	Translation-buffer disable (2)
25	CDR	Cache disable
26	MCESR	Machine-check error summary
27	CAER	Cache error (2)
30	SBIFS	SBI fault status (2)
31	SBIS	SBI silo (2)
32	SBISC	SBI silo comparator (2)
33	SBIMT	SBI maintenance (2)
34	SBIER	SBI error (2)
35	SBITA	SBI timeout address (2)
36	SBIQC	SBI quadword clear (2)
37	IORESET	I/O reset
3B	TBDATA	Translation-buffer data
3C	MBRK	Microprogram breakpoint
3D	PME	Performance-monitor enable
3E	SID	System identification
3F	TBCHK	Translation-buffer check (3)

(1) subset implementation
(2) reads as zero, ignores writes
(3) always returns "TB miss"

```
 3             2 2           1 1 1
 1             4 3           7 6 5             8 7             0
+---------------+-------------+-+-+---------------+---------------+
|       7       |   reserved  |D| microcode rev | hardware rev  |
+---------------+-------------+-+-+---------------+---------------+
```

Figure B-3 MicroVAX I System Identification Register (SID)

```
+----------------------------------------------------------------+
|                    byte count (0000000C hex)                   | :SP
+----------------------------------------------------------------+
|                     machine-check type code                    |
+----------------------------------------------------------------+
|                        first parameter                         |
+----------------------------------------------------------------+
|                        second parameter                        |
+----------------------------------------------------------------+
|                               PC                               |
+----------------------------------------------------------------+
|                               PSL                              |
+----------------------------------------------------------------+
```

Figure B-4 MicroVAX I Machine-Check Stack Frame

Table B-4: MicroVAX I Machine-Check Type Codes

Code	Meaning
0	Memory-controller bug check (1)
1	Unrecoverable memory-read error (1)
2	Nonexistent memory (1)
3	Illegal I/O-space operation (1)
4	Unrecoverable PTE-read error (1)
5	Unrecoverable PTE-write error (1)
6	Control-store parity error (2)
7	Micromachine bug check (2)
8	Q22-bus vector read error (2)
9	Write parameter error (3)

(1) Bits<29,21:0> of the first parameter contain the corresponding bits of the physical address of the last memory reference, and the second parameter contains the address presented to the memory controller.

(2) Both parameters are zero.

(3) The first parameter contains the virtual address that was being written, and the second parameter is zero.

```
                    +--------------------------------+
0000 0000:          |                                |
                    |        installed memory        |
                    |                                |
    :               | - - - - - - - - - - - - - - -  |
                    |                                |
                    |      memory address space      |
                    |     beyond installed memory    |
00FF FFFF:          |                                |
                    +--------------------------------+
0100 0000:          |                                |
                    |            reserved            |
1FFF FFFF:          |                                |
                    +--------------------------------+
2000 0000:          |                                |
                    |        Q22-bus I/O space       |
2000 1FFF:          |                                |
                    +--------------------------------+
2000 2000:          |                                |
                    |            reserved            |
2003 FFFF:          |                                |
                    +--------------------------------+
2004 0000:          |                                |
                    |            ROM space           |
2007 FFFF:          |                                |
                    +--------------------------------+
2008 0000:          |                                |
                    |    local register I/O space    |
200B FFFF:          |                                |
                    +--------------------------------+
200C 0000:          |                                |
                    |            reserved            |
2FFF FFFF:          |                                |
                    +--------------------------------+
3000 0000:          |                                |
                    |      Q22-bus memory space      |
303F FFFF:          |                                |
                    +--------------------------------+
3040 0000:          |                                |
                    |            reserved            |
3FFF FFFF:          |                                |
                    +--------------------------------+
```

Figure B-5 MicroVAX II Physical Address Space

Table B-5: MicroVAX II Implementation-Dependent SCB Vectors

Offset	Vector Name	IPL	Notes
C0	Interval timer	16	
200 - 3FC	Q22-bus interrupts	14-17	IPL corresponds to bus request levels 4 through 7.

Table B-6: MicroVAX II Implementation-Dependent IPRs

IPR	Mnemonic	Name
18	ICCS	Interval-clock control and status (1)
19	NICR	Next interval count (2)
1A	ICR	Interval count (2)
1B	TODR	Time-of-year clock (2)
1C	CSRS	Console storage receiver status (2)
1D	CSRD	Console storage receiver data (2)
1E	CSTS	Console storage transmitter status (2)
1F	CSTD	Console storage transmitter data (2)
20	RXCS	Console receiver status (2)
21	RXDB	Console receiver data (2)
22	TXCS	Console transmitter status (2)
23	TXDB	Console transmitter data (2)
24	TBDR	Translation-buffer disable (2)
25	CADR	Cache disable (2)
26	MCESR	Machine-check error summary (2)
27	CAER	Cache error (2)
29	SAVISP	Console saved interrupt stack pointer
2A	SAVPC	Console saved PC
2B	SAVPSL	Console saved PSL
30	SBIFS	SBI fault status (2)
31	SBIS	SBI silo (2)
32	SBISC	SBI silo comparator (2)
33	SBIMT	SBI maintenance (2)
34	SBIER	SBI error (2)
35	SBITA	SBI timeout address (2)
36	SBIQC	SBI quadword clear (2)
37	IORESET	I/O reset (2)
3B	TBDATA	Translation-buffer data (2)
3C	MBRK	Microprogram breakpoint (2)
3D	PME	Performance-monitor enable (2)
3E	SID	System identification
3F	TBCHK	Translation-buffer check

(1) Subset implementation
(2) Reads as zero, ignores writes

```
 3             2 2
 1             4 3                                               0
+--------------+-------------------------------------------------+
|      8       |                        0                        |
+--------------+-------------------------------------------------+
```

Figure B-6 MicroVAX II System Identification Register (SID)

```
+----------------------------------------------------------------+
|                  byte count (0000000C hex)                     | :SP
+----------------------------------------------------------------+
|                      machine-check code                        |
+----------------------------------------------------------------+
|                 most recent virtual address                    |
+----------------------------------------------------------------+
|                 internal state information                     |
+----------------------------------------------------------------+
|                              PC                                |
+----------------------------------------------------------------+
|                              PSL                               |
+----------------------------------------------------------------+
```

Figure B-7 MicroVAX II Machine-Check Stack Frame

Table B-7: MicroVAX II Machine-Check Type Codes

Code	Meaning
1	Impossible microcode state (FSD)
2	Impossible microcode state (SSD)
3	Undefined FPU error code 0
4	Undefined FPU error code 7
5	Undefined memory management status (TB miss)
6	Undefined memory management status (M = 0)
7	Process PTE in P0 space
8	Process PTE in Pl space
9	Undefined interrupt ID code
80	Read bus error, address parameter is virtual
81	Read bus error, address parameter is physical
82	Write bus error, address parameter is virtual
83	Write bus error, address parameter is physical

Table B-8: MicroVAX II Halt Codes

Code	Meaning
2	HALT L asserted
3	Initial power on
4	Interrupt stack not valid during exception
5	Machine check during machine check or kernel-stack-not-valid exception
6	HALT instruction executed in kernel mode
7	SCB vector bits <1:0> = 11
8	SCB vector bits <1:0> = 10
A	Change-mode from the interrupt stack
10	Access-control violation or translation not valid during machine check
11	Access-control violation or translation not valid during kernel-stack-not-valid exception

```
                 +--------------------------------+
0000 0000:       |                                |
                 |        installed memory        |
                 |                                |
    :            | - - - - - - - - - - - - - - -  |
                 |                                |
                 |     memory address space       |
                 |    beyond installed memory     |
                 |                                |
00EF FFFF:       |                                |
                 +--------------------------------+
00F0 0000:       |                                |
                 |            reserved            |
00F1 FFFF:       |                                |
                 +--------------------------------+
00F2 0000:       |     memory adapter space       |
                 +--------------------------------+
00F2 2000:       |    reserved adapter space      |
                 +--------------------------------+
00F2 4000:       |    reserved adapter space      |
                 +--------------------------------+
00F2 6000:       |     UNIBUS adapter space       |
                 +--------------------------------+
00F2 8000:       |    reserved adapter space      |
    :            |               :                |
00F2 FFFF:       |    reserved adapter space      |
                 +--------------------------------+
00F3 0000:       |                                |
                 |                                |
                 |                                |
                 |            reserved            |
                 |                                |
                 |                                |
00FB FFFF:       |                                |
                 +--------------------------------+
00FC 0000:       |                                |
                 |                                |
                 |     UNIBUS address space       |
                 |                                |
00FF FFFF:       |                                |
                 +--------------------------------+
```

Figure B-8 VAX-11/730 Physical Address Space

Table B-9: VAX-11/730 Implementation-Dependent SCB Vectors

Offset	Vector Name	IPL	Notes
54	Corrected read data	1A	Corrected memory error.
F0	Console storage device receive	14	Console load device signaling read complete.
F4	Console storage device transmit	14	Console load device signaling write complete.
200 - 3FC	UNIBUS interrupts	14-17	IPL corresponds to bus-request levels 4 through 7.

Table B-10: VAX-11/730 Halt Codes

Code	Meaning
?02	CTRL/P was typed at the console.
03	Does not appear in a halt message, but is passed by the console during powerfail restart.
?04	The interrupt stack was not valid when the processor tried to push PC and PSL from an exception or an interrupt.
?05	While the processor was trying to process a machine-check, a second machine-check occurred.
?06	A HALT instruction was executed while the processor was in kernel mode.
?07	An exception or interrupt occurred and the SCB vector had bit<1> set.
?0A	A CHMx instruction was executed when the processor was executing on the interrupt stack.
?0B	A CHMx instruction was executed and the SCB vector had bit<0> set.
?0C	A hard memory error occurred while the processor was trying to read an SCB vector.

Table B-11: VAX-11/730 Implementation-Dependent IPRs

IPR	Mnemonic	Name
1C	CSRS	Console storage receive status
1D	CSRD	Console storage receive data
1E	CSTS	Console storage transmit status
1F	CSTD	Console storage transmit data
24	TBDR	Translation-buffer disable (1)
25	CDR	Cache disable (1)
26	MCESR	Machine-check error summary (2)
27	CAER	Cache error (1)
28	ACCS	Accelerator control and status
30	SBIFS	SBI fault status (1)
31	SBIS	SBI silo (3)
32	SBISC	SBI silo comparator (1)
33	SBIMT	SBI maintenance (1)
34	SBIER	SBI error (1)
35	SBITA	SBI timeout address (3)
36	SBIQC	SBI quadword clear (4)
37	IORESET	I/O reset
3D	PME	Performance-monitor enable
3E	SID	System identification
3F	TBCHK	Translation-buffer check

(1) Reads as zero, ignores writes
(2) Reads as zero, any write clears the "machine-check in progress" flag
(3) Reads as zero, writes cause reserved-operand fault
(4) Ignores writes, reads cause reserved-operand fault

```
 3             2 2             1 1
 1             4 3             6 5             8 7             0
+---------------+---------------+---------------+---------------+
|       3       |   reserved    | microcode rev |   reserved    |
+---------------+---------------+---------------+---------------+
```

Figure B-9 VAX-11/730 System Identification Register (SID)

```
+--------------------------------------------------------------+
|                  byte count (0000000C hex)                   | :SP
+--------------------------------------------------------------+
|                   machine-check type code                    |
+--------------------------------------------------------------+
|                       first parameter                        |
+--------------------------------------------------------------+
|                       second parameter                       |
+--------------------------------------------------------------+
|                              PC                              |
+--------------------------------------------------------------+
|                              PSL                             |
+--------------------------------------------------------------+
```

Figure B-10 VAX-11/730 Machine-Check Stack Frame

Table B-12: VAX-11/730 Machine-Check Type Codes

Code	Meaning
0	Microcode shouldn't be here. If the first parameter is zero, no other information is available. If the first parameter is two, the problem was inability to write back a PTE<M> bit. If the parameter is three, the problem was a bad 8085 interrupt. The second parameter is always zero.
1	Translation-buffer parity error. The first parameter is the bad value from the TB. PFN is in bits <23:0>. PTE<V>, the protection code, and PTE<M> are in bits <31:26>. TB valid bit is in bit <25>. The second parameter is the virtual address referenced.
3	Impossible value in memory CSR. The first parameter is the virtual address referenced. The second parameter is the bad value of the CSR.
4	Fast interrupt without support. A fast interrupt was requested and no microcode was loaded to handle it. Both parameters are zero.
5	FPA parity error. The FPA control store had a parity error. The first parameter has parity error summary in bit<0>, group 0 parity in bit<1>, group 1 parity in bit<2>, and in unpredictable in bits<31:3>. The second parameter is zero.
6	Error on SPTE read. The first parameter is the physical address of the SPTE. The second parameter contains the error syndrome bits.
7	Uncorrectable ECC error. The first parameter is the physical address of the reference. The second parameter contains the error syndrome bits.
8	Nonexistent memory. The first parameter is the physical address referenced. The second parameter is zero.
9	Unaligned or non-longword reference to I/O space. The first parameter is the physical address referenced. The second parameter is zero.
A	Illegal I/O-space address. The first parameter is the physical address referenced. The second parameter is zero.
B	Illegal UNIBUS reference. The first parameter is the physical address referenced. The second parameter is zero.

```
                +--------------------------------+
0000 0000:      |       installed memory         |
                | - - - - - - - - - - - - - - -  |
    :           |     memory address space       |
                |    beyond installed memory     |
00EF FFFF:      |                                |
                +--------------------------------+
00F0 0000:      |   reserved for loading WCS     |
                +--------------------------------+
00F2 0000:      |     memory adapter space       |
                +--------------------------------+
00F2 2000:      |    reserved adapter space      |
                +--------------------------------+
00F2 4000:      |    reserved adapter space      |
                +--------------------------------+
00F2 6000:      |    reserved adapter space      |
                +--------------------------------+
00F2 8000:      |    MASSBUS 0 adapter space     |
                +--------------------------------+
00F2 A000:      |    MASSBUS 1 adapter space     |
                +--------------------------------+
00F2 C000:      |    MASSBUS 2 adapter space     |
                +--------------------------------+
00F2 E000:      |    MASSBUS 3 adapter space     |
                +--------------------------------+
00F3 0000:      |    UNIBUS 0 adapter space      |
                +--------------------------------+
00F3 2000:      |    UNIBUS 1 adapter space      |
                +--------------------------------+
00F3 4000:      |                                |
    :           |    reserved adapter spaces     |
00F3 E000:      |                                |
                +--------------------------------+
00F4 0000:      |                                |
    :           |           reserved             |
00F7 FFFF:      |                                |
                +--------------------------------+
00F8 0000:      |                                |
    :           |     UNIBUS 1 address space     |
00FB FFFF:      |                                |
                +--------------------------------+
00FC 0000:      |                                |
    :           |     UNIBUS 0 address space     |
00FF FFFF:      |                                |
                +--------------------------------+
```

Figure B-11 VAX-11/750 Physical Address Space

Table B-13: VAX-11/750 Implementation-Dependent SCB Vectors

Offset	Vector Name	IPL	Notes
54	Corrected read data, or read data substitute	1A	Corrected memory error, or uncorrected memory error.
60	Write bus error	1D	Taken regardless of current IPL if error occurs during exception or interrupt.
F0	Console storage device (TU58) receive	17	Console load device signaling read complete.
F4	Console storage device (TU58) transmit	17	Console load device signaling write complete.
100 - 13C	Adapter interrupts, adapters 0 through 15	14	Adapter interrupt.
140 - 17C	Adapter interrupts, adapters 0 through 15	15	Adapter interrupt.
180 - 1BC	Adapter interrupts, adapters 0 through 15	16	Adapter interrupt.
1C0 - 1FC	Adapter interrupts, adapters 0 through 15	17	Adapter interrupt.
200 - 3FC	UNIBUS interrupts	14-17	IPL corresponds to bus request levels 4 through 7.

Table B-14: VAX-11/750 Halt Codes

Code	Meaning
1	Successful completion of console TEST command
2	Processor halted by ^P or single step
3	Power up
4	Interrupt stack not valid, or SCB read failure
5	Double bus write error
6	HALT instruction executed
7	Illegal interrupt or exception vector (bits<1:0> = 3)
8	Jump to nonexistent user writable control store (SCB vector bits<1:0> = 2, and no user WCS installed)
A	Change-mode from the interrupt stack
B	Change-mode to the interrupt stack
11	Can't find a valid Restart Parameter Block during power-up restart, and power-up action switch set to RESTART/HALT
12	"System restart in progress" flag already set during power-up restart, and power-up action switch set to RESTART/HALT
13	Can't find 64k bytes of good memory during system bootstrap
14	Bad boot ROM or no boot ROM during power-up bootstrap
15	"System bootstrap in progress" flag already set during boot
16	Power up and power-up action switch set to HALT
FF	Self test failure

Table B-15: VAX-11/750 Implementation-Dependent IPRs

IPR	Mnemonic	Name
17	CMIERR	CMI error
1C	CSRS	Console storage receive status
1D	CSRD	Console storage receive data
1E	CSTS	Console storage transmit status
1F	CSTD	Console storage transmit data
24	TBDR	Translation-buffer disable
25	CADR	Cache disable
26	MCESR	Machine-check error summary
27	CAER	Cache error
28	ACCS	Accelerator control and status
37	IORESET	Initialize UNIBUS interconnect
3B	TB	Translation-buffer test
3D	PME	Performance-monitor enable
3E	SID	System identification
3F	TBCHK	Translation-buffer check

```
 3              2 2             1 1
 1              4 3             6 5            8 7             0
+---------------+---------------+---------------+---------------+
|       2       |   reserved    | microcode rev | hardware rev  |
+---------------+---------------+---------------+---------------+
```

Figure B-12 VAX-11/750 System Identification Register (SID)

```
+-----------------------------------------------------------------+
| count of bytes pushed, excluding PC, PSL, and count.  28 hex.  | :SP
+-----------------------------------------------------------------+
|                          error code                             |
+-----------------------------------------------------------------+
|                          VA register                            |
+-----------------------------------------------------------------+
|                  PC at the time of the error                    |
+-----------------------------------------------------------------+
|                              MDR                                |
+-----------------------------------------------------------------+
|                      saved mode register                        |
+-----------------------------------------------------------------+
|                       read lock timeout                         |
+-----------------------------------------------------------------+
|                TB group parity error register                   |
+-----------------------------------------------------------------+
|                     cache error register                        |
+-----------------------------------------------------------------+
|                      bus error register                         |
+-----------------------------------------------------------------+
|             machine-check error summary register                |
+-----------------------------------------------------------------+
|                               PC                                |
+-----------------------------------------------------------------+
|                               PSL                               |
+-----------------------------------------------------------------+
```

Figure B-13 VAX-11/750 Machine-Check Stack Frame

Table B-16: VAX-11/750 Machine-Check Error Summary Register

Code	Meaning
1	Control-store parity error
2	Translation-buffer parity error, bus error, or cache parity error
6	"Microcode shouldn't be here" error
7	"Unused IRD ROM slot" error

```
            +--------------------------------+
0000 0000:  |                                |
            |        installed memory        |
            |                                |
    :       | - - - - - - - - - - - - - - -  |
            |                                |
            |      memory address space      |
            |    beyond installed memory     |
1FFF FFFF:  |                                |
            +--------------------------------+
2000 0000:  |        TR0 adapter space       |
            +--------------------------------+
2000 2000:  |        TR1 adapter space       |
            |                                |
    :       |                :               |
            |                                |
2001 E000:  |        TR15 adapter space      |
            +--------------------------------+
2002 0000:  |                                |
            |                                |
    :       |            reserved            |
            |                                |
200F FFFF:  |                                |
            +--------------------------------+
2010 0000:  |     UNIBUS 0 address space     |
            +--------------------------------+
2014 0000:  |     UNIBUS 1 address space     |
            +--------------------------------+
2018 0000:  |     UNIBUS 2 address space     |
            +--------------------------------+
2018 0000:  |     UNIBUS 3 address space     |
            +--------------------------------+
2020 0000:  |                                |
            |                                |
    :       |            reserved            |
            |                                |
3FFF FFFF:  |                                |
            +--------------------------------+
```

Figure B-14 VAX-11/780 Physical Address Space

Table B-17: VAX-11/780 Implementation-Dependent SCB Vectors

Offset	Vector Name	IPL	Notes
50	SBI silo compare	19	System-bus error.
54	Corrected read data, or read data substitute	1A	Corrected memory error, or uncorrected memory error.
58	SBI alert	1B	System-bus error.
5C	SBI fault	1C	System-bus error.
60	Memory write timeout	1D	Memory error.
100-13C	Nexus interrupts, nexuses 0 through 15	14	Device or adapter interrupt.
140-17C	Nexus interrupts, nexuses 0 through 15	15	Device or adapter interrupt.
180-1BC	Nexus interrupts, nexuses 0 through 15	16	Device or adapter interrupt.
1C0-1FC	Nexus interrupts, nexuses 0 through 15	17	Device or adapter interrupt.

Table B-18: VAX-11/780 Halt Codes

Code	Message	Meaning
3	none	Power up.
4	?INT-STK INVLD	The interrupt stack was not valid when the processor attempted to take an exception or interrupt.
5	?CPU DBLE-ERR HLT	A second processor error occurred during the processing of a previous error.
7	?ILL I/E VEC	Illegal interrupt or exception vector. (Vector bits<1:0> were 3.)
8	?NO USR WCS	Jump to nonexistent user writable control store.
0A	?CHM ERR	Change-mode from the interrupt stack.
0F		Clock Phase Error Halt (VAX-11/785).

Table B-19: VAX-11/780 Implementation-Dependent IPRs

IPR	Mnemonic	Name
20	RXCS	Console terminal receive control and status
21	RXDB	Console terminal receive data buffer
22	TXCS	Console terminal transmit control and status
23	TXDB	Console terminal transmit data buffer
28	ACCS	Accelerator control and status
29	ACCR	Accelerator maintenance
2C	WCSA	Writable-control-store address
2D	WCSD	Writable-control-store data
30	SBIFS	SBI fault status
31	SBIS	SBI silo
32	SBISC	SBI silo comparator
33	SBIMT	SBI maintenance
34	SBIER	SBI error
35	SBITA	SBI timeout address
36	SBIQC	SBI quadword clear
3C	MBRK	Microprogram breakpoint
3D	PME	Performance-monitor enable
3E	SID	System identification
3F	TBCHK	Translation-buffer check

```
 3             2 2 2             1 1   1 1
 1             4 3 2             5 4   2 1                    0
+---------------+-+-+---------------+-----+----------------------+
|       1       | | |   ECO level   |plant|    serial number     |
+---------------+-+-+---------------+-----+----------------------+
                 |
                 0 = VAX-11/780
                 1 = VAX-11/785
```

Figure B-15 VAX-11/780 System Identification Register (SID)

```
+-----------------------------------------------------------------+
| count of bytes pushed, excluding PC, PSL, and count.  28 hex. | :SP
+-----------------------------------------------------------------+
|                      summary parameter                          |
+-----------------------------------------------------------------+
|                  CPU error status register                      |
+-----------------------------------------------------------------+
|                       trapped microPC                           |
+-----------------------------------------------------------------+
|                         VA or VIBA                              |
+-----------------------------------------------------------------+
|                         D register                              |
+-----------------------------------------------------------------+
|                      TB ERR 0 register                          |
+-----------------------------------------------------------------+
|                      TB ERR 1 register                          |
+-----------------------------------------------------------------+
|                       timeout address                           |
+-----------------------------------------------------------------+
|                       parity register                           |
+-----------------------------------------------------------------+
|                      SBI error register                         |
+-----------------------------------------------------------------+
|                             PC                                  |
+-----------------------------------------------------------------+
|                             PSL                                 |
+-----------------------------------------------------------------+
```

Figure B-16 VAX-11/780 Machine-Check Stack Frame

Table B-20: VAX-11/780 Machine-Check Error Summary Parameter

Code	Meaning
00	Central-processor read timeout or error confirmation fault
02	Central-processor translation-buffer parity error fault
03	Central-processor cache parity error fault
05	Central-processor read data substitute fault
0A	Instruction-buffer translation-buffer parity error fault
0C	Instruction-buffer read data substitute fault
0D	Instruction-buffer read timeout or error confirmation fault
0F	Instruction-buffer cache parity error fault
F1	Control-store parity error abort
F2	Central-processor translation-buffer parity error abort
F3	Central-processor cache parity error abort
F4	Central-processor read timeout or error confirmation abort
F5	Central-processor read data substitute abort
F6	"Microcode not supposed to get here" abort

```
                +--------------------------------+
0000 0000:      |        installed memory        |
    :           | - - - - - - - - - - - - - - -  |
                |      memory address space      |
1FFF FFFF:      |    beyond installed memory     |
                +--------------------------------+
2000 0000:      |        node 0 nodespace        |
                +--------------------------------+
2000 2000:      |        node 1 nodespace        |
    :           |               :                |
2001 FFFF:      |        node 15 nodespace       |
                +--------------------------------+
2002 0000:      |                                |
    :           |            reserved            |
2003 FFFF:      |                                |
                +--------------------------------+
2004 0000:      |                                |
    :           |       node private space       |
203F FFFF:      |                                |
                +--------------------------------+
2040 0000:      |             node 0             |
                |          window space          |
                +--------------------------------+
2044 0000:      |             node 1             |
                |          window space          |
    :           |               :                |
                |             node 15            |
207F FFFF:      |          window space          |
                +--------------------------------+
2080 0000:      |                                |
    :           |            reserved            |
3FFF FFFF:      |                                |
                +--------------------------------+
```

Figure B-17 VAX-11/8200 Physical Address Space

Table B-21: VAX-11/8200 Implementation-Dependent SCB Vectors

Offset	Vector Name	IPL
50	BI bus-error interrupt	14
54	Corrected read data	1A
58	RXCD (receive data register)	14
80	Interprocessor interrupt	14
C0	Interval timer interrupt	16
C8	Serial line #1 RX interrupt	14
CC	Serial line #1 TX interrupt	14
D0	Serial line #2 RX interrupt	14
D4	Serial line #2 TX interrupt	14
D8	Serial line #3 RX interrupt	14
DC	Serial line #3 TX interrupt	14
F0	Console storage device	14
F8	Console terminal RX interrupt	14
FC	Console terminal TX interrupt	14
100-3FFC	BI defined, loaded by software	14-17

Table B-22: VAX 8200 Implementation-Dependent IPRs

IPR	Mnemonic	Name
16	IPIR	Interprocesssor interrupt request
20	RXCS	Console terminal receive control and status
21	RXDB	Console terminal receive data buffer
22	TXCS	Console terminal transmit control and status
23	TXDB	Console terminal transmit data buffer
24	TBDR	Translation-buffer disable
25	CADR	Cache disable
26	MCESR	Machine-check error summary
28	ACCS	Accelerator control and status
2C	WCSA	Writable-control-store address
2D	WCSD	Writable-control-store data
2E	WCSL	Writable-control-store load
3E	SID	System identification
50	RXCS1	Serial line 1 receive control and status
51	RXDB1	Serial line 1 receive data buffer
52	TXCS1	Serial line 1 transmit control and status
53	TXDB1	Serial line 1 transmit data buffer
54	RXCS2	Serial line 2 receive control and status
55	RXDB2	Serial line 2 receive data buffer
56	TXCS2	Serial line 2 transmit control and status
57	TXDB2	Serial line 2 transmit data buffer
58	RXCS2	Serial line 3 receive control and status
59	RXDB3	Serial line 3 receive data buffer
5A	TXCS3	Serial line 3 transmit control and status
5B	TXDB3	Serial line 3 transmit data buffer
5C	RXCD	Receive console data
5D	CACHEX	Cache invalidate
5E	BINID	BI node identification
5F	BISTOP	BI stop

```
 3             2 2 2               1 1
 1             4 3 2               9 8            9 8   7         0
+--------------+-+-----------------+-------------+-+------------+
|      5       | | CPU revision    |  patch rev  |1| ucode rev  |
+--------------+-+-----------------+-------------+-+------------+
                |
                0 = VAX 8200 or VAX 8300
                1 = VAX 8250 or VAX 8350
```

Figure B-18 VAX 8200 System Identification Register (SID)

```
+-----------------------------------------------------------------+
| count of bytes pushed, excluding PC, PSL, and count.  20 hex. | :SP
+-----------------------------------------------------------------+
|                    machine-check type code                      |
+-----------------------------------------------------------------+
|                          parameter 1                            |
+-----------------------------------------------------------------+
|                              VA                                 |
+-----------------------------------------------------------------+
|                           VA prime                              |
+-----------------------------------------------------------------+
|                             MAR                                 |
+-----------------------------------------------------------------+
|                          status word                            |
+-----------------------------------------------------------------+
|                        PC at failure                            |
+-----------------------------------------------------------------+
|                     micro-PC at failure                         |
+-----------------------------------------------------------------+
|                              PC                                 |
+-----------------------------------------------------------------+
|                             PSL                                 |
+-----------------------------------------------------------------+
```

Figure B-19 VAX 8200 Machine-Check Stack Frame

```
               +--------------------------------+
0000 0000:     |        installed memory        |
    :          | - - - - - - - - - - - - - - -  |
1FFF FFFF:     |   nonexistent-memory space     |
               +--------------------------------+
2000 0000:     |    SBI0.TR0 adapter space      |
               +--------------------------------+
2000 2000:     |    SBI0.TR1 adapter space      |
    :          |              :                 |
2001 FFFF:     |    SBI0.TR15 adapter space     |
               +--------------------------------+
2002 0000:     |           reserved             |
               +--------------------------------+
2008 0000:     |       SBIA #0 registers        |
               +--------------------------------+
2008 00C0:     |           reserved             |
               +--------------------------------+
2010 0000:     |    UNIBUS 0 address space      |
               +--------------------------------+
2014 0000:     |    UNIBUS 1 address space      |
               |              :                 |
201F FFFF:     |    UNIBUS 3 address space      |
               +--------------------------------+
2020 0000:     |           reserved             |
               +--------------------------------+
2200 0000:     |    SBI1.TR0 adapter space      |
               +--------------------------------+
2200 2000:     |    SBI1.TR1 adapter space      |
    :          |              :                 |
2201 FFFF:     |    SBI1.TR15 adapter space     |
               +--------------------------------+
2202 0000:     |           reserved             |
               +--------------------------------+
2208 0000:     |       SBIA #1 registers        |
               +--------------------------------+
2208 00C0:     |           reserved             |
               +--------------------------------+
2210 0000:     |    UNIBUS 4 address space      |
               +--------------------------------+
2214 0000:     |    UNIBUS 5 address space      |
               |              :                 |
221F FFFF:     |    UNIBUS 7 address space      |
               +--------------------------------+
2220 0000:     |           reserved             |
               +--------------------------------+
2400 0000:     |     SBIA #2 address space      |
               +--------------------------------+
2600 0000:     |     SBIA #3 address space      |
               +--------------------------------+
2800 0000:     |           reserved             |
               +--------------------------------+
```

Figure B-20 VAX-11/8600 Physical Address Space

Table B-23: VAX-11/8600 Implementation-Dependent SCB Vectors

Offset	Vector Name	IPL	Notes
04	machine checks	1D or 1F	At IPL 1D only if the error is unrelated to the current instruction.
50	SBI0 silo compare	19	System-bus error.
54	Corrected read data	1D	Corrected memory error.
58	SBI0 alert	1B	System-bus error.
5C	SBI0 fault	1C	System-bus error.
60	SBIA0 internal error	1D	Abus-to-SBI-adapter error.
64	SBI0 power fail	1E	
100-13C	SBIA0 nexus interrupts at BR4, nexus 0 thru 15	14	Device or adapter interrupt.
140-17C	SBIA0 nexus interrupts at BR5, nexus 0 thru 15	15	Device or adapter interrupt.
180-1BC	SBIA0 nexus interrupts at BR6, nexus 0 thru 15	16	Device or adapter interrupt.
1C0-1FC	SBIA0 nexus interrupts at BR7, nexus 0 thru 15	17	Device or adapter interrupt.
250	SBI1 silo compare	19	System-bus error.
258	SBI1 alert	1B	System-bus error.
25C	SBI1 fault	1C	System-bus error.
260	SBIA1 internal error	1D	Abus-to-SBI-adapter error.
264	SBI1 power fail	1E	
300-33C	SBIA1 nexus interrupts at BR4, nexus 0 thru 15	14	Device or adapter interrupt.
340-37C	SBIA1 nexus interrupts at BR5, nexus 0 thru 15	15	Device or adapter interrupt.
380-3BC	SBIA1 nexus interrupts at BR6, nexus 0 thru 15	16	Device or adapter interrupt.
3C0-3FC	SBIA1 nexus interrupts at BR7, nexus 0 thru 15	17	Device or adapter interrupt.
400-5FC	IOA #2 vectors		Correspond to 200-3FC.
600-7FC	IOA #3 vectors		Correspond to 200-3FC.

Table B-24: VAX 8600 Halt Codes

Code	Message	Meaning
0	UNRECOVERABLE MACHINE HANG	Console-support microcode is not running.
4	INTERRUPT STACK INVALID	Interrupt stack not valid during the initiation of an exception or interrupt.
5	NON-EBOX DOUBLE ERROR	While initiating a machine check, a second machine check occurred.
6	KERNEL MODE HALT	HALT instruction in kernel mode.
7	SCB VECTOR<1:0>=3, INVALID	Illegal SCB vector (bits<1:0>=3).
8	SCB VECTOR<1:0>=2, NO USER WCS	Illegal SCB vector (bits<1:0>=2, no WCS microcode).
9	ERROR PENDING ON HALT	Pending error on HALT.
A	CHM WITH IS=1	CHMx from the interrupt stack.
B	CHM WITH VECTOR<1:0> NOT 0	CHMx to the interrupt stack.
11	INVOKED BY CONSOLE	Operator typed HALT at console.

Table B-25: VAX 8600 Implementation-Dependent IPRs

IPR	Mnemonic	Name
20	RXCS	Console terminal receive control and status
21	RXDB	Console terminal receive data buffer
22	TXCS	Console terminal transmit control and status
23	TXDB	Console terminal transmit data buffer
28	ACCS	Accelerator control and status
3D	PME	Performance-monitor enable
3E	SID	System identification
3F	TBCHK	Translation-buffer check
40	PAMACC	Physical address memory map access
41	PAMLOC	Physical address memory map location
42	CSWP	Cache sweep
43	MDECC	M-box data ECC
44	MENA	M-box error enable
45	MDCTL	M-box data control
46	MCCTL	M-box MCC control
47	MERG	M-box error generator
48	CRBT	Console reboot
49	DFI	Diagnostic fault insertion
4A	EHSR	Error handling status
4C	STXCS	Console block storage control and status
4D	STXDB	Console block storage data buffer
4E	ESPA	E-box scratchpad address
4F	ESPD	E-box scratchpad data

```
 3             2 2 2              1 1   1 1                       
 1             4 3 2              6 5   2 1                      0
+--------------+-+---------------+-----+-----------------------+
|      4       | |   ECO level   |plant|     serial number     |
+--------------+-+---------------+-----+-----------------------+
                |
                0 = VAX 8600
                1 = VAX 8650
```

Figure B-21 VAX 8600 System Identification Register (SID)

```
+-----------------------------------------------------------------+
| count of bytes pushed, excluding PC, PSL, and count.  58 hex.   |
+-----------------------------------------------------------------+
|                            EHM.STS                              |
+-----------------------------------------------------------------+
|                            EVMQSAV                              |
+-----------------------------------------------------------------+
|                             EBCS                                |
+-----------------------------------------------------------------+
|                            EDPSR                                |
+-----------------------------------------------------------------+
|                            CSLINT                               |
+-----------------------------------------------------------------+
|                            IBESR                                |
+-----------------------------------------------------------------+
|                            EBXWD1                               |
+-----------------------------------------------------------------+
|                            EBXWD2                               |
+-----------------------------------------------------------------+
|                            IVASAV                               |
+-----------------------------------------------------------------+
|                            VIBASAV                              |
+-----------------------------------------------------------------+
|                            ESASAV                               |
+-----------------------------------------------------------------+
|                            ISASAV                               |
+-----------------------------------------------------------------+
|                             CPC                                 |
+-----------------------------------------------------------------+
|                            MSTAT1                               |
+-----------------------------------------------------------------+
|                            MSTAT2                               |
+-----------------------------------------------------------------+
|                            MDECC                                |
+-----------------------------------------------------------------+
|                             MERG                                |
+-----------------------------------------------------------------+
|                            CSHCTL                               |
+-----------------------------------------------------------------+
|                             MEAR                                |
+-----------------------------------------------------------------+
|                             MEDR                                |
+-----------------------------------------------------------------+
|                            FBXERR                               |
+-----------------------------------------------------------------+
|                             CSES                                |
+-----------------------------------------------------------------+
|                              PC                                 |
+-----------------------------------------------------------------+
|                              PSL                                |
+-----------------------------------------------------------------+
```

Figure B-22 VAX 8600 Machine-Check Stack Frame

Table B-26: VAX 8600 Machine-Check Stack Frame Contents

Field	Offset	Extent	Meaning
COUNT	00	<31:0>	Bytes pushed, excluding PC, PSL, and count.
EHM.STS	04	<31:24>	Error-handling status.
		<23:19>	Control-store correction request.
		<15:8>	Trap vector.
		<7:0>	Status code.
EVMQSAV	08	<31:0>	E-box virtual address.
EBCS	0C	<31:27>	E-box control-store parity error.
		<15:8>	E-box, I-box, M-box error.
		<4:0>	Abort flags.
EDPSR	10	<31:24>	A-mux and B-mux byte in error.
		<15:12>	VMQ byte in error.
		<11:0>	E-box datapath error flags.
CSLINT	14	<29:23>	Interrupt request flags.
		<22:21>	IOA number.
		<20:16>	Interrupt priority requests.
		<15:0>	C-bus data, control, and address.
IBESR	18	<31:21>	I-box error flags.
		<15:8>	Diagnostic and maintenance flags.
EBXWD1	1C	<31:0>	Top of scratch-pad stack.
EBXWD2	20	<31:0>	Next on scratch-pad stack.
IVASAV	24	<31:0>	Virtual address for operand fetch.
VIBASAV	28	<31:0>	Virtual address of next IB port request to fill IB.
ESASAV	2C	<31:0>	PC being evaluated by E-box.
ISASAV	30	<31:0>	PC being evaluated by operand fetch unit.
CPC	34	<31:0>	PC being evaluated by I-buffer.
MSTAT1	38	<31:26>	M-box cycle in error.
		<25:24>	Word count.
		<23:16>	M-box error conditions.
		<15:12>	Cache hit/miss history.
		<11:8>	TB errors.
		<7:0>	M-box datapath error summary.
MSTAT2	3C	<20:16>	PAMM data.
		<15:8>	SBIA diagnostic status.
		<7:0>	M-box error information.
MDECC	40	<22:16>	Data ECC error flags.
		<14:9>	Data ECC syndrome.
		<7:1>	Data ECC check bit invert.
		<0>	Longword parity invert.
MERG	44	<12:9>	Diagnostic bits.
		<8>	Memory management enable.
		<5:0>	M-box diagnostic error-insertion bits.
CSHCTL	48	<3:0>	Cache control.
MEAR	4C	<29:2>	Physical address latched.
MEDR	50	<31:00>	Data word latched.
FBXERR	54	<25:9>	Accelerator status.
CSES	58	<28:16>	Control-store address.
		<15:8>	Control-store syndrome.
		<2:0>	Control-store code.
PC	5C	<31:0>	PC.
PSL	60	<31:0>	PSL.

```
                    +---------------------------------+
0000 0000:          |        installed memory         |
    :               | - - - - - - - - - - - - - - - - |
                    |     memory address space        |
1FFF FFFF:          |    beyond installed memory      |
                    +---------------------------------+
2000 0000:          |     BI#0 node 0 nodespace       |
                    +---------------------------------+
2000 2000:          |     BI#0 node 1 nodespace       |
    :               |               :                 |
2001 FFFF:          |     BI#0 node 15 nodespace      |
                    +---------------------------------+
2002 0000:          |        multicast space          |
                    +---------------------------------+
2004 0000:          |           boot ROM              |
                    +---------------------------------+
2006 0000:          |           reserved              |
                    +---------------------------------+
2008 0000:          |       node private space        |
                    +---------------------------------+
2010 0000:          |           reserved              |
                    +---------------------------------+
2040 0000:          |      node 0 window space        |
                    +---------------------------------+
2044 0000:          |      node 1 window space        |
    :               |               :                 |
207F FFFF:          |      node 15 window space       |
                    +---------------------------------+
2080 0000:          |           reserved              |
                    +---------------------------------+
2200 0000:          |          BI #1 space            |
                    +---------------------------------+
2400 0000:          |          BI #2 space            |
                    +---------------------------------+
2600 0000:          |          BI #3 space            |
                    +---------------------------------+
2800 0000:          |                                 |
    :               |           reserved              |
3FFF FFFF:          |                                 |
                    +---------------------------------+
```

Figure B-23 VAX-11/8800 Physical Address Space

Table B-27: VAX-11/8800 Implementation-Dependent SCB Vectors

Offset	Vector Name	IPL	Notes
5C	NMI fault	1C	System-bus error.
80	interprocessor interrupt	14	Not included in 8500.
148	memory error	15	Corrected or uncorrected error, interlock timeout, or controller error.
100-3FFC	BI defined, loaded by software	14-17	See VAX 8800 BI Vectors (VMS V4.4), below.

Table B-28: VAX 8800 BI Vectors (VMS V4.4)

Offset	Vector Name	IPL	Notes
100-13C	NMI0 nexus interrupts at BR4, nexus 0 thru 15	14	Device or adapter interrupt.
140-17C	NMI0 nexus interrupts at BR5, nexus 0 thru 15	15	Device or adapter interrupt.
180-1BC	NMI0 nexus interrupts at BR6, nexus 0 thru 15	16	Device or adapter interrupt.
1C0-1FC	NMI0 nexus interrupts at BR7, nexus 0 thru 15	17	Device or adapter interrupt.
200-3FC	UNIBUS device interrupts	14-17	Devices on UNIBUS 0.
400-5FC	UNIBUS device interrupts	14-17	Devices on UNIBUS 1.
600-38FC			Unused.
3800			Passive Release.
3804-38FF			Unused.
3900-393C	BI#0 interrupts, nodes 0 through 15	14	Device or adapter interrupt.
3940-397C	BI#0 interrupts, nodes 0 through 15	15	Device or adapter interrupt.
3980-39BC	BI#0 interrupts, nodes 0 through 15	16	Device or adapter interrupt.
39C0-39FC	BI#0 interrupts, nodes 0 through 15	17	Device or adapter interrupt.
3A00			Passive Release.
3A04-3AFF			Unused.
3B00-3BFC	BI#1 interrupt vectors	14-17	Device or adapter interrupt.
3C00			Passive Release.
3C04-3CFF			Unused.
3D00-3DFC	BI#2 interrupt vectors	14-17	Device or adapter interrupt.
3E00			Passive Release.
3E00-3EFF			Unused.
3F00-3FFC	BI#3 interrupt vectors	14-17	Device or adapter interrupt.

Table B-29: VAX 8800 Implementation-Dependent IPRs

IPR	Mnemonic	Name
26	MCSTS	Machine check status
80	NICTRL	NMI interrupt control
81	INOP	Interrupt other processor
82	NMIFSR	NMI fault/status
83	NMISILO	NMI bus silo
84	NMIEAR	NMI error address
85	COR	Cache on
86	REVR1	Revision register #1
87	REVR2	Revision register #2
88	CLRTOSTS	Clear timeout status

```
 3             2 2 2           1 1
 1             4 3 2           6 5                           0
+---------------+-+---------------+-----------------------------+
|       6       | | CPU revision  |        serial number        |
+---------------+-+---------------+-----------------------------+
                 |
                 0 = Right processor
                 1 = Left processor
```

Figure B-24 VAX 8800 System Identification Register (SID)

```
+------------------------------------------------------------+
| count of bytes pushed, excluding PC, PSL, and count.  20 hex. |   :SP
+------------------------------------------------------------+
|                           MCSTS                            |
+------------------------------------------------------------+
|                             PC                             |
+------------------------------------------------------------+
|                          VA/VIBA                           |
+------------------------------------------------------------+
|                            IBER                            |
+------------------------------------------------------------+
|                            CBER                            |
+------------------------------------------------------------+
|                            EBER                            |
+------------------------------------------------------------+
|                           NMIFSR                           |
+------------------------------------------------------------+
|                           NMIEAR                           |
+------------------------------------------------------------+
|                             PC                             |
+------------------------------------------------------------+
|                            PSL                             |
+------------------------------------------------------------+
```

Figure B-25 VAX 8800 Machine-Check Stack Frame

Table B-30: VAX 8800 Machine-Check Stack Frame Contents

Mnemonic	Offset	Contents
COUNT	00	Count of bytes pushed, excluding PC, PSL, and count.
MCSTS	04	Machine-check status.
PC	08	Current PC.
VA/VIBA	0C	Virtual address/virtual instruction-buffer address.
IBER	10	IBOX error.
CBER	14	CBOX error.
EBER	18	EBOX error.
NMIFSR	1C	NMI fault summary.
MNIEAR	20	NMI error address.
PC	24	PC of faulted opcode.
PSL	28	Processor status longword.

```
                    +--------------------------------+
 0000 0000:         |                                |
                    |      Read/write memory         |
                    |                                |
                    | - - - - - - - - - - - - - - -  |
                    |                                |
                    |          I/O space             |
                    |                                |
                    | - - - - - - - - - - - - - - -  |
                    |                                |
                    |      Nonexistent memory        |
                    |   (memory address space        |
                    |   beyond installed memory)     |
 3FFF FFFF:         |                                |
                    +--------------------------------+
```

Figure B-26 VVAX Physical Address Space

Table B-31: VVAX Implementation-Dependent SCB Vectors

Offset	Vector Name	IPL	Notes
100 - 1FC			Unused
200 - 204			Reserved
208	disk controller	15	
20C			Reserved
210	terminal controller	15	Receive interrupt
214	terminal controller	15	Transmit interrupt
218	tape controller	15	
21C			Reserved
220	printer controller	15	
224 - 3FC			Reserved

Table B-32: VVAX Implementation-Dependent IPRs

IPR	Mnemonic	Name
18	ICCS	Interval clock control and status (subset implementation)
20	RXCS	console receive control and status
21	RXDB	console receive data buffer
22	TXCS	console transmit control and status
23	TXDB	console transmit data buffer
26	MCESR	machine check error summary
28	ACCS	accelerator control and status (ignores writes, reads 0)
37	IORESET	I/O adapter reset
3D	PME	performance monitor enable
3E	SID	system identification
3F	TBCHK	translation buffer check
64	MEMSIZE	physical memory size
65	KCALL	kernel call

```
 3              2 2             1 1
 1              4 3             6 5                            0
+---------------+---------------+-------------------------------+
|       9       |   VVAX type   |           ECO level           |
+---------------+---------------+-------------------------------+
```

Figure B-27 VVAX System Identification Register (SID)

Table B-33: Fields of the VVAX SID Register

Extent	Name	Meaning
<31:24>	CPU type	Contains the value 9, which indicates that the processor is a VVAX
<23:16>	VVAX type	Identifies the major version number of the VVAX implementation.
<15:0>	ECO level	Indicates the minor version number of the VVAX implementation.

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

Figure B-28 VVAX MEMSIZE Register

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

Figure B-29 VVAX KCALL Register

```
+---------------------------------------------------------+
|  count of bytes pushed, excluding PC, PSL, and count.   |  :SP
|              equals 0C hex (12 decimal).                |
+---------------------------------------------------------+
|                 machine check type code                 |
+---------------------------------------------------------+
|                     first parameter                     |
+---------------------------------------------------------+
|                     second parameter                    |
+---------------------------------------------------------+
|                           PC                            |
+---------------------------------------------------------+
|                           PSL                           |
+---------------------------------------------------------+
```

Figure B-30 VVAX Machine-Check Stack Frame

Table B-34: VVAX Machine-Check Type Code Parameter

Code	Meaning
00	Uncorrectable ECC error. The first parameter is the physical address of the reference. The second parameter contains the error syndrome bits.
01	Nonexistent memory. The first parameter is the physical address referenced. The second parameter is zero.
02	Access violation on physical memory. An attempt was made to write into a valid, read-only memory location. The first parameter is the physical address referenced. The second parameter is zero.

Table B-35: VVAX Implementation-Dependent Halt Codes

Code	Meaning
02	Control-P was typed at the VVAX processor's console.
03	Does not appear in a halt message, but is passed by the console during powerfail restart.
04	The interrupt stack was not valid when the processor tried to push PC, PSL, or a parameter from an exception or an interrupt.
05	A second machine check occurred while the processor was processing a previous machine check.
06	A HALT instruction was executed while the processor was in kernel mode.
07	An exception or interrupt occurred and the SCB vector had bit <1> set.
0A	A CHMx instruction was executed when the processor was executing on the interrupt stack (PSL<IS> was set).
0B	A CHMx instruction was executed and the SCB vector had bit <0> set.
0C	An uncorrectable ECC error occurred while the VVAX processor was trying to read an exception or interrupt vector from the SCB.
12	Restart-in-progress flag already set during restart.
13	Could not find 64K bytes of good memory during system bootstrap.
15	Boot-in-progress flag already set during boot.
17	VMB.EXE could not be found on the boot device, or there was an error while loading VMB.EXE.
18	Software attempted to set an SLR or SBR value that would make system space not fit entirely in VM-physical memory while memory management was enabled. This can occur with an MTPR to MAPEN, SLR, or SBR.
19	Software attempted to make the size of process space larger than the Max_process_space_per_VM SYSGEN parameter while memory management was enabled. This can occur in an LDPCTX, or in an MTPR to MAPEN, P0LR, or P1LR.
1A	Software attempted to place the P0 page table in a location outside of system space while memory management was enabled. This can occur in an LDPCTX, or in an MTPR to MAPEN, P0BR, P0LR, or SLR.
1B	Software attempted to place the P1 page table in a location outside of system space while memory management was enabled. This can occur in an LDPCTX, or in an MTPR to MAPEN, P1BR, P1LR, or SLR.
1C	The VVAX processor attempted to reference a process PTE, and it was not accessible.
1D	Software attempted to reference non-existent memory.
1E	Software attempted to set SLR to a value larger than the Max_system_space_per_VM SYSGEN parameter while memory management was enabled. This can occur either with an MTPR to SLR or to MAPEN.

NOTE

\With the vast number of VAX processor configurations that are now available, the VAX Architecture group no longer has the resources to update this appendix. The major problem is acquiring the implementation specific information to put here. In some cases, this information is not even written down in one place. Complicating this situation is the varying combinations of processors and platforms that are now possible.

While we will no longer research new products to update this appendix, we will update the appendix with submissions from others. Any submission will be gratefully accepted. Send them to EAGLE1::SRM.\

Change History:

Revision J. Rich Brunner, December 1989.

- o There's too few of us and too many of them. Someone else has to supply new processor information for this appendix.
- o Expanded list of UNPREDICTABLES. It became so large it had to be removed from the SRM and made into a separate document.
- o Expanded list of UNDEFINED.

Revision H. Tim Leonard, May 1987.

- o Describe VVAX.
- o Rewrite the description of physical memory, to allow 34-bit physical addresses.

Revision F. Al Thomas, November 1986.

- o Change the descriptions of MicroVAX I and MicroVAX II to account for the new Implementation Options.

Revision E. Al Thomas, September 1986.

- o Add sections for 8200, 8300, 8500, 8650, and 8800.

Revision D. Tim Leonard, March 1985.

- o Change the revision number to correspond to DEC Standard 032 rev number.
- o Rename to Appendix B.
- o Change the organization from by-feature to by-processor.
- o Add sections for MicroVAX I, VAX-11/785, and VAX 8600.
- o Fill in the section for MicroVAX I.
- o Update the section on BI I/O space.
- o Page tables, PCB and SCB must not be in I/O space.
- o Emulation can use stack space.

Revision 1. Tim Leonard, 25 September 1982.

APPENDIX C

ARCHITECTURE MANAGEMENT

This appendix describes the architecture-management process for specifications managed by the Systems Architecture group, and has two purposes. The first is to point out responsibilities for adherence to, interpretation of, and changes to architecture specifications. The second is to define a formal ECO (engineering change order) process that includes a careful review of the impact that a change or exception has on existing and planned products based on the architecture. The goal is to promote hardware and software product compatibility.

The process described here applies to all specifications managed by the Systems Architecture group. The final section lists these specifications, as well as the current maintainer (called "architect" in the rest of this document) for each specification.

C.1 RESPONSIBILITIES

Ensuring product compatibility takes effort on the part of managers, engineers, the architecture group, and architecture-review-group representatives. Their responsibilities in the process are described here.

C.1.1 Project, Product, and Engineering Managers

For each architecture specification that applies to a product, managers of the hardware and software design and support organizations must:

- o Ensure that their engineers are aware of the existence of the architecture specification and the long-term adverse consequences to Digital of deviations from it.

- o Designate a representative to vote on architecture issues and to transmit information about the architecture between the organization and the architecture review group (ARG).

- Develop and successfully carry out a plan to demonstrate conformance of their implementation to the architecture specification.

- Provide resources to fix, and a schedule to release fixes for, all non-waived deviations from the architecture.

C.1.2 Hardware, Microcode, and Software Engineers

Hardware, microcode, and software engineers must be familiar with, and implement designs that conform to, the architecture specification. Hardware and microcode engineers must ensure that ANY software that conforms will execute correctly with their hardware. Similarly, software engineers must ensure that ANY hardware that conforms will work correctly with their software. The software must not depend on any hardware that handles unpredictable or undefined architecture features. All nonconformities must be fixed by product support people or waived by the Systems Architecture group using the ECO process.

C.1.3 Systems Architecture Group and Architects

The Systems Architecture group (and in particular the architect of each of the specifications the group maintains) interprets and develops the specifications, as well as manages the process for review and approval of ECOs to them. More specifically, the group:

- Interprets and clarifies the architecture specifications. This may result in editorial changes that in most cases will not require a formal ECO proposal and vote. The architect publishes and distributes such changes in the same way that approved ECO proposals are made known.

- Decides on the organizations to be represented in architecture review groups, with the goal of including all technical views pertaining to the architecture. The architect asks organization management to choose representatives and considers requests from organizations to participate.

- Processes ECO proposals, including the gathering of all appropriate comment, building consensus, forming the final ECO for inclusion in a specification, collecting votes, and achieving closure on the proposal.

- Processes exceptions to the architecture specifications. These waiver requests are processed as ECO proposals.

- Archives and disseminates the specifications. This includes keeping the sources, incorporating and distributing changes, maintaining change history data, and updating distribution lists.

- o Monitors for compliance and develops architecture verification tools for compliance monitoring.
- o Maintains and publishes the lists of known architecture discrepancies.

C.1.4 Architecture-Review-Group Representatives

An ARG representative serves as an organization's primary contact with the architecture group. ARG members are responsible for:

- o Circulating ECO proposals and results, and other information sent to them concerning the architecture, to all projects in the organization they represent.
- o Collecting and returning all comments and their vote on ECO proposals to the architect by the deadline stated in each proposal, unless an extension is agreed to by the architect.
- o Notifying the architect of the latest functional and design specifications and reviews for implementations within the organization they represent.
- o Reporting and helping to resolve discrepancies between implementations within their organization and the architecture.

C.2 SPECIFICATION DISTRIBUTION

Due to the confidential nature of architecture specifications, the Systems Architecture group will follow these guidelines when distributing copies:

- o Requests from non-Digital employees, whether or not non-disclosure agreements have been signed, will be reviewed on a case-by-case basis by the Systems Architecture manager and possibly others in the management chain. Only hard copy will be sent, and the architect is responsible for keeping a list of where and to whom copies were sent.
- o Information-only requests from Digital employees will be answered by providing a hard copy of the specification and information on how to find out about updates. Copies will be sent to Digital mailstops only, and the architect is responsible for keeping a list of where and to whom copies were sent.

- Requests from Digital employees who are members of the architecture review group, or who are working on a project with close ties to the specification, will be answered by providing hard or soft copy at their preference.

Requested copies of architecture specifications must not be distributed further.

C.3 CONFORMANCE AND WAIVERS

Any plan to test implementation conformance to an architecture specification must include the following criteria:

- Correct operation with current existing implementations, based on product requirements for coexistence of, and cooperation between, the current and new implementations.

- Passing any established verification procedure that is specified by the Systems Architecture group.

The group responsible for each implementation is required to develop a plan to demonstrate conformance to the architecture. This verification plan must be reviewed and approved by the Systems Architecture group, and then successfully carried out on the version of the product planned for first customer ship. For each version of the product released after first customer ship, this verification plan must be carried out successfully as well. Those assigned to do the testing must notify the Systems Architecture group in writing when this work has been completed, including details on what was tested (including versions) and the results.

When any discrepancy between a design or implementation and the architecture is discovered, the group's ARG member must bring the issue to the attention of the architect before any hardware or software design or implementation decisions are committed. Each implementation must conform to the architecture specification unless it has been granted a waiver by the ECO process. A waiver may allow an immediate shipment followed by a fix. It may require some instances to be fixed and not others, or it may allow a permanent exception. No "system killer" will ever be given more than a temporary waiver with a very short duration. Digital makes fixes for non-waived deviations available to customers. To minimize the need for ECOs and waivers, the group's ARG member should notify the architect of the existence of, and reviews for, functional and design specifications as early as possible. The architect, or someone working for the architect and not a member of the design or support team, can work to clarify and resolve issues before they require formal ECO resolution.

C.3.1 Publishing Architecture Discrepancies

The Systems Architecture group maintains the lists of known architecture discrepancies. These lists are made available to ARG members for distribution within their organizations. The lists are submitted to publications that are readily accessible to customers if the discrepancies are visible to customers. All discrepancies that have been waived will be published in the architecture specification, either as a note in an appropriate section, or in an appendix for waivers.

The single exception is that we will not publish the list of "system killers" outside of Digital. All questions about "system killers", even ones asking if there are any, will be answered "No Comment". The reason for this is to protect customers (from their users) by providing absolutely no information on the subject. In addition, this "no comment" policy will be published along with the lists of architecture discrepancies.

C.4 ECO PROCESS

The ECO process is biased against change. Architecture changes or exceptions that cause incompatible differences between hardware implementations, or cause software not to run on any hardware that conforms to the architecture, require compelling reasons.

C.4.1 Participants in the ECO Process

The following people and groups are participants in the architecture review process:

- o The architect, who manages the review process for the specification. The architect reports to the Systems Architecture manager in this capacity.
- o The ARG consisting of the architect, former major contributors to the specification, and representatives of software and hardware engineering groups that are strongly dependent on the architecture specification. Managers of the engineering groups select these representatives.
- o The manager of the Systems Architecture group, who participates in ECO approval and appeal.
- o Other reviewers, as deemed appropriate by ECO process participants.

C.4.2 ECO Proposals

Proposers of an ECO must submit an ECO proposal in writing to the architect for the specification. Proposing an ECO includes whatever preliminary investigation is necessary to achieve a sound technical proposal. The authors must detail the rationale and known consequences. Among these are:

- Function changes.
- The performance gain (or loss).
- Any effects on existing and planned implementations.
- The plan to change prior implementations, including estimated costs, or the plan to deal with incompatibilities if prior implementations are not changed.
- A hardware and software transition plan.
- A description and cost estimate of any other hardware, software, or economic impact.

An ECO proposal does not always require all of the above information. ARG members must provide information on the impact of the proposed change for the projects they represent as part of the ECO review process.

Once submitted, the architect works with the originators of a proposal to:

- Ensure there is information in the proposal about who to contact for more details on the issue.
- Include alternative solutions, with their pros and cons, in the proposal.
- Present the proposed alternative in final form, and in the context of the architecture specification. This includes exact wording changes and additions.

C.4.3 ECO Proposal Review

The architect logs, assigns a number, and mails an ECO proposal to the following reviewers for comments and their vote by the stated deadline:

- Members of the ARG, on whom the technical integrity and continuity of the architecture depends.

- o Others directly involved in the technical issue.
- o Whoever else is deemed appropriate by reviewers.

The architect keeps the list of ARG members and the complete distribution list for any ECO proposal.

The reviewers are responsible for circulating the ECO proposal within the organization they represent, collecting the comments, and returning the comments in writing to the architect by the stated deadline for comments and voting. The architect collects the comments and makes them available to other reviewers. This may result in individual discussions, meetings, and revisions to the ECO proposal. Any non-editoral revision of an ECO proposal must be sent to all reviewers for comments and voting again.

There are ECO proposals that contain sensitive information. As a security precaution, all notes conferences that are used for discussion of ECO proposals are private. Only ARG members, or individuals sponsored by ARG members, are given access to these conferences.

The reviewers must submit their vote (YES, NO, or ABSTAIN) in writing to the architect by the deadline stated in a proposal for comments and voting. ARG members voting for a proposal are asked to explain the impact on their organization and how it would be dealt with. ARG members voting against a proposal are asked to explain their objections and suggest alternatives to the proposal that would be acceptable. The architect tabulates the votes, and attempts to build a consensus based on sound judgment, not just expedient compromise.

C.4.4 ECO Approval And Appeal

To pass, an ECO must have the approval of the Systems Architecture manager and the consensus of the reviewers. The architect determines if a consensus exists using these guidelines:

- o A quorum consisting of at least 80% of the reviewers must submit a YES or NO vote. Votes to ABSTAIN or no response from more than 20% of the reviewers requires follow-up before a consensus can be determined.
- o At least 75% of the reviewers in the quorum must vote YES.
- o Any strong opposition is an indication of no consensus.

If an ECO proposal is about to pass and there were NO votes, those casting NO votes will be given an opportunity to accept or reject the consensus.

An architect's decision about an ECO proposal can be appealed to the Systems Architecture manager and further up the management chain if necessary. In an appeal, management works to find a compromise that is acceptable to the architect, the author(s) of the ECO proposal, those objecting to the architect's decision, and all reviewers.

C.4.5 Publishing And Distribution of ECO Results

The architect notifies the ARG about the decision on an ECO proposal in a timely manner. Background data on this decision, including the text of votes submitted by reviewers, is available to ARG members upon request. When approved, the architect incorporates the ECO into the specification with change bars where appropriate and possible, and makes change pages available to ARG members. ARG members are responsible for making known the results to the projects they represent.

C.4.6 Specification Retirement

The architecture specification will be considered retired when a proposal that it be abandoned meets with the same approval as required for ECO proposals and exceptions.

C.5 SYSTEMS-ARCHITECTURE-GROUP SPECIFICATIONS

Document	Architect
DEC Standard 032 (VAX Architecture Standard)	Rich Brunner
DEC Standard 057 (VAXBI Standard)	Rich Best
XMI Standard	Rich Best
BI VAX Port Architecture	Bob Chen

Last Update: December 1989

Revision History:

Revision J. Rich Brunner, December 1989.

- o Updated list of systems-architecture specifications.

Revision 1. C. Wiecek, 13 February 1986.

- o Version for approval

Revision 2. C. Wiecek, 27 March 1986.

- o Added one arch. group responsibility
- o Put ARG responsibilities together in a section
- o Made clarifications to implementation conformance, publishing system killers, ECO proposal review, determining consensus, and publishing ECO results

Revision 3. C. Wiecek, 14 April 1986.

- o Process put into effect
- o Minor edits and clarification to ECO appeal

Revision 4. C. Wiecek, 30 September 1986.

- o Added notes-conference access policy in ECO Proposal Review section
- o Named new CI VAX Port architect
- o Minor edits

Revision 5. C. Wiecek, 26 May 1987.

- o Loosened wording on fixes for non-waived deviations made by Digital to more closely match current policy
- o Updated Systems Architecture Group list of specifications
- o Changed document format to an appendix

digital™

INDEX

abort, 5-3
  (see also exceptions)
absolute queue
  data type, 1-12
  definition of, 3-95
  instructions, 3-97
ACB - add compare and branch instructions, 3-49
access control (see memory, protection of)
access mode, 1-26, 4-13
  changing of, 4-33
  in a vector processor, 13-23, 14-3, 14-36
  in a virtual machine, 12-16
  use of, in protection of memory, 4-16
access type, 4-13
  indication of, 4-32
  modify, 4-8, 4-32
  notation for, 3-2
  use of, in protection of memory, 4-8, 4-13
  write, 4-32
access-control-violation fault, 4-16, 5-12
ACCVIO (see access-control-violation fault)
ACV (see access-control-violation fault)
ADAWI - add aligned word interlocked instruction, 3-7, 7-6 to 7-8, 7-25
ADC - add carry instructions, 9-19
ADD - add instructions, 9-27
  ADAWI - add aligned word interlocked, 3-7, 7-6 to 7-8, 7-25
  ADWC - add with carry, 3-9
  floating point, 3-121
  floating point, vector, 13-45
  integer, 3-8
  integer, vector, 13-33
  packed decimal, 3-187
address, 1-3
address access type, 3-2
  (see also access type)
address instructions, 3-37
address space number register (see ASN)
address translation, 4-8
  in a vector processor, 14-41
  process space, 4-21 to 4-22, 4-26
  rtVAX, 11-5 to 11-6
  system space, 4-16
  translation buffer, 4-27
  when mapping is disabled, 4-6
addressing modes
  base operand specifier, 2-14
  branch mode addressing, 2-3, 2-15
  general mode addressing, 2-3, 2-7 to 2-10, 2-13 to 2-15
ADWC - add with carry instruction, 3-9
alignment
  of branch destinations, 3-49
  of I/O registers, 7-23
  of stacks, 5-28
  of vectors, 4-32, 13-23, 14-16
AOBLEQ - add one and branch less than or equal instruction, 3-51
AOBLSS - add one and branch less than instruction, 3-52
AP - argument pointer, 1-22
architecture
  definition of, xi
  lists of implementation bugs, xiii
  testing conformance, xi
argument pointer (AP), 1-22
argument validation, 4-33
ASH - arithmetic shift instructions
  compatibility mode, 9-36
  integer, 3-10
  packed decimal, 3-189
ASHC - arithmetic shift combined instruction, 9-37
ASL - arithmetic shift left instructions, 9-18
ASN - address space number register, 4-5, 4-28, 6-5, 6-9, 11-9
  in the PCB, 6-4
ASR - arithmetic shift right instructions, 9-17
assembler notation, vector, 13-65
AST - asynchronous system trap, 5-38, 6-5
  ASTLVL - AST level register, 6-5 to 6-6
  software interrupt, 6-7
ASTLVL - AST level register, 6-6

asynchronous memory management exception handling, 14-2, 14-16
asynchronous system trap (see AST)
at console command (@ console command), 10-28
atomic operands, 7-5
atomic operations, 7-1
  changing page table entries, 4-12
  modify operands, 2-19
  on I/O device registers, 7-23
  update of PTE modify, 4-11
atomic references, 7-4
autorestart, 10-7
AXE testing, xi

B - conditional branch instructions, 3-53, 9-40
base operand specifier, 2-14
base register, 1-22
BB - branch on bit instructions
  and modify interlocked, 3-58, 7-6 to 7-8, 7-23
  and modify without interlock, 3-56
  branch on low bit, 3-60
  non-interlocked, 3-55
BI interconnect (see VAXBI)
BIC - bit clear instructions, 9-34
  BICPSW - bit clear PSW, 3-81
  logical, 3-11
BICPSW - bit clear PSW instruction, 3-81
BIS - bit set instructions, 9-33
  BISPSW - bit set PSW, 3-82
  logical, 3-12
BISPSW - bit set PSW instruction, 3-82
BIT - bit test instructions, 3-13, 9-35
bit field, 3-41
  access type, 3-2
  data type, 1-10
  FIELD addressing notation, 3-41
  in registers, 1-23
BLB - branch on low bit instructions, 3-60
BOOT console command, 10-14
bootstrap, 10-1
  restart algorithm, 10-4
  RPB - restart parameter block, 10-5
  the bootstrap algorithm, 10-2
  the console BOOT command, 10-14
BPT - breakpoint instruction, 3-83
BPT - breakpoint trap instruction, 9-6, 9-54
BR - branch instructions
  BRB, BRW, 3-61
  compatibility mode, 9-39
branch displacement access type, 3-2
  (see also access type)
branch mode addressing, 2-15
  definition of, 2-3
breakpoint fault, 5-15
BSB - branch to subroutine instructions, 3-62
BUG - bugcheck instructions, 3-84
bugcheck, 3-84
bugs
  lists of implementation bugs, xiii
byte
  in registers, 1-22
  notation for, 3-3
byte data type
  definition of, 1-3
byte-within-page, 4-4

C - carry condition code, 1-27
caches
  restrictions on, 7-17 to 7-18
CALL - call instructions
  CALLG, 3-73
  CALLS, 3-75
call frame, 3-71
carry condition code (C), 1-27
CASE - case instructions, 3-63
CC - condition code operator instructions, 9-51
CCC - clear all condition codes instruction, 9-51
chaining of vector instructions, 14-6
change mode
  exception, 5-15
  instructions, 5-40
  use of the instructions, 4-33
character string
  data type, 1-17
  in registers, 1-23
  instructions, 3-151
CHM - change mode instructions, 5-40
  use of, 4-33
CLC - clear C condition code instruction, 9-51
CLN - clear N condition code instruction, 9-51
clock

  interval timer, 8-22, 11-4
  time-of-year, 8-22, 11-4
CLR - clear instructions, 9-11
  floating point, 3-123
  integer and logical, 3-14
CLV - clear V condition code instruction, 9-51
CLZ - clear Z condition code instruction, 9-51
CM - compatibility mode, 1-26
CMP - compare instructions, 9-29
  character string, 3-153
  floating point, 3-124
  floating point, vector, 13-47
  integer and logical, 3-15
  integer, vector, 13-35
  packed decimal, 3-191
  variable-length bit field, 3-42
COM - complement instructions, 9-16
command files (@ console command), 10-28
compatibility mode, 9-1
  address modes, 9-2
  addresses, 9-53
  bit in the PSL (CM), 1-26
  entering into, 9-53
  exceptions, 5-14, 9-54
  exiting from, 9-53
  general registers, 9-2
  I/O, 9-57
  instructions, 9-6
  interrupts, 9-54
  memory management, 9-53
  omission of, 11-4
  processor registers, 9-57
  PSW - processor status word, 9-6
  register mapping, 9-53
  stack, 9-6
  synchronization, 9-57
  tracing, 9-55
  unimplemented traps, 9-56
condition codes, 1-23
console, 10-10
  console I/O mode, 10-1, 10-10
  program I/O mode, 10-1
  registers, 10-33
@ console command, 10-28
context of a process, 6-1
context switching, scalar, 5-5 to 5-6, 14-2, 14-4, 14-35
context switching, vector, 14-3, 14-21
CONTINUE console command, 10-14
control characters as console commands, 10-11
control instructions, 3-49
control region of process space, 4-2
control variable, interlock, 7-6 to 7-7
corruption, of data, 7-4
CPUID - CPU identification register, 8-17
CRC - calculate cyclic redundancy check instruction, 3-178
CSM - call supervisor mode instruction, 9-6
CTRL/C console command, 10-12
CTRL/O console command, 10-12
CTRL/P console command, 10-12
CTRL/Q console command, 10-12
CTRL/S console command, 10-12
CTRL/U console command, 10-11
CUR_MOD - current mode, 1-26
current mode (CUR_MOD), 1-26
CVT - convert instructions
  CVTLP - convert long to packed, 3-193
  CVTPL - convert packed to long, 3-194
  CVTPS - convert packed to separate numeric, 3-196
  CVTPT - convert packed to trailing numeric, 3-198
  CVTSP - convert separate numeric to packed, 3-200
  CVTTP - convert trailing numeric to packed, 3-202
  floating point, 3-125
  integer, 3-16
  vector, 13-49

D_floating
  data type, 1-7
  in registers, 1-23
  notation for, 3-3
data types
  definitions of, 1-3, 1-6 to 1-7, 1-9 to 1-10, 1-12, 1-17 to 1-18
  in registers, 1-22
  notation for, 3-2
data, shared
  atomicity of, 7-4 to 7-5
  corruption of, 7-4, 7-6
  granularity of, 7-5
  interlocked access, 7-6
  mutual exclusion, 7-1, 7-6
  visibility, 7-2
DEC - decrement instructions, 9-12
  integer, 3-17
DEC Standard 032
  abstract of, i

  applicability of, i
  definition of, i
  for customers, ii
  how to order, ii
  manager responsible for, ii
  revision history of, ii
  status of, i
decimal overflow trap
  enable bit (DV), 1-27
decimal string
  data types, 1-17 to 1-18
  divide-by-zero trap, 5-11
  in registers, 1-23
  instructions, 3-183
  overflow trap, 5-11
default result, vector arithmetic exceptions, 13-4, 13-43, 14-19
dependences, vector results, 14-10
DEPOSIT console command, 10-15
disabled fault, vector processor, 14-19, 14-21
DIV - divide instruction, 9-31
DIV - divide instructions
  EDIV - extended divide, 3-20
  floating point, 3-129
  floating point, vector, 13-51
  integer, 3-18
  packed decimal, 3-204
divide-by-zero
  condition in vector processor, 13-10, 13-43
  fault, 5-12
  trap, 5-11
double precision floating point (see D_floating and G_floating)
DV - decimal overflow trap enable, 1-27

edit instructions, vector, 13-57
EDITPC - edit packed to character string instruction, 3-216
EDIV - extended divide instruction, 3-20
EMOD - extended multiply and integerize instructions, 3-131
EMT - emulator trap instruction, 9-6, 9-54
EMUL - extended multiply instruction, 3-21
emulation
  exception, 5-14, 11-11, 11-13
  of VM instructions, 12-13
entry mask, 3-71
EO$_ - EDITPC pattern operators
  EO$ADJUST_INPUT, 3-223
  EO$BLANK_ZERO, 3-224
  EO$CLEAR_SIGNIF, 3-234
  EO$END, 3-225
  EO$END_FLOAT, 3-226
  EO$FILL, 3-227
  EO$FLOAT, 3-228
  EO$INSERT, 3-230
  EO$LOAD_FILL, 3-231
  EO$LOAD_MINUS, 3-231
  EO$LOAD_PLUS, 3-231
  EO$LOAD_SIGN, 3-231
  EO$MOVE, 3-232
  EO$REPLACE_SIGN, 3-233
  EO$SET_SIGNIF, 3-234
  EO$STORE_SIGN, 3-235
errors, 5-4
  serious system failures, 5-18
ESP - executive stack pointer, 5-27
  in the PCB, 6-4
ETYPE, 13-10
ETYPE field for vector exceptions, 13-44
EXAMINE console command, 10-20
exceptions, 5-3, 5-13, 5-15
  arithmetic exceptions, 5-11
  compatibility mode exceptions, 5-14
  definition of, 5-1 to 5-2
  emulation exceptions, 5-14, 11-11, 11-13
  initiation of, 5-31
  memory management exceptions, 5-12
  precedence of, 2-19, 5-12, 5-15, 5-21
  restrictions to allow restarting instructions, 5-5
  serious system failures, 5-18
  vectors in the SCB, 5-23, 5-28
exceptions, vector processor, 13-13, 14-15, 14-26
  memory management (see memory management exceptions, vector)
  arithmetic, 13-4, 13-43, 14-7, 14-15, 14-19
  floating-point, 13-43
execution model, vector processor, 14-1
executive mode, 1-26
  VM-emulation trap referencing, 4-35
executive stack, 5-27
  (see stack and stack pointer)

EXT - extract field instructions, 3-43
extended instructions, 9-6

F_floating
  data type, 1-7
  in registers, 1-22
  notation for, 3-3
FADD - floating point add instruction, 9-6
fault
  parameters (see stack frame)
  (see also exceptions)
  definition of, 5-3
  precedence of, 4-16, 4-30
FDIV - floating point divide instruction, 9-6
FF - find first bit instructions, 3-44
field (see bit field)
FIND console command, 10-22
first part done (FPD), 1-26
floating point
  data types, 1-7, 1-9, 1-22 to 1-23
  divide-by-zero condition in vector processor, 13-10, 13-43
  divide-by-zero fault, 5-12
  in literal addressing mode, 2-13
  instructions, 3-115
  overflow condition in vector processor, 13-10, 13-43
  overflow fault, 5-12
  reserved operand condition, 3-116, 5-13, 13-10, 13-43
  underflow condition in vector processor, 13-10, 13-43
  underflow fault, 3-117, 5-12
  vector instructions, 13-43
floating underflow exception
  enable bit (FU), 1-27
FMUL - floating point multiply instruction, 9-6
FP - frame pointer, 1-22
FP11 floating point instructions, 9-6
FPD - first part done, 1-26
frame pointer (FP), 1-22
FSUB - floating point subtract instruction, 9-6
FU - floating underflow exception enable, 1-27

G_floating
  data type, 1-9
  in registers, 1-23
  notation for, 3-3
general mode addressing, 2-7
  definition of, 2-3
general purpose registers (see registers)
general registers (see registers)
global page, 7-19
GPRs - general purpose registers (see registers)
granularity of data accesses, 7-5

H_floating
  data type, 1-9
  in registers, 1-23
  notation for, 3-3
halt
  console command, 10-22
  effect on vector processor, 14-36
  halt codes, 10-31
  instruction, 3-85
  interrupt stack not valid, 5-19
  synchronizing vector memory before, 14-35
HALT - halt instruction, 3-85, 9-6
hardware errors, vector, 14-20, 14-43

I/O, 7-18
  I/O address space, B-2
  I/O registers, 7-23
  instructions usable to reference I/O space, 7-24
  interlocked instructions accessing I/O space, 7-8
  PTEs for I/O devices, 7-19
  restrictions on caches, 7-18
  restrictions on references to I/O space, 4-32, 5-5, 14-16, 14-35 to 14-36, 14-42
ICCS - interval clock control and status register, 8-22 to 8-23, 11-4
ICR - interval count register, 8-22 to 8-23, 11-4
immediate mode addressing, 2-8
  restrictions in vector processor, 13-25, 13-27
implementation options, 11-1
  address space number (ASN), 11-9
  application extension groups, 11-3
  base instruction group, 11-2
  compatibility mode, 11-4
  emulated-only instruction group, 11-3

extended-accuracy group, 11-3
instruction set, 11-1
internal processor registers, 11-4
memory management options, 11-4
packed-decimal-string group, 11-3
PDP-11 compatibility mode, 11-4
rtVAX, 11-5
UNIBUS interconnect, 11-9
vector processor, 11-9
virtual-machine support, 11-4
INC - increment instructions, 9-13
integer, 3-22
incoherency, of data, 7-4
INDEX - compute index instruction, 3-86
index register, 1-22
initialization, effects of, 10-7
INITIALIZE console command, 10-23
initiate exception or interrupt, 5-31
INS - insert queue instructions
INSQHI - insert at head interlocked, 3-97, 7-6 to 7-7, 7-15
INSQTI - insert at tail interlocked, 3-100, 7-6 to 7-7
INSQUE - insert, 3-103
instruction buffer
flushing by REI, 5-38, 7-10
instruction format, 2-1
instruction interpretation
by hardware, 2-18, 5-22, 7-9
by software, 5-14, 11-11, 11-13
for vector instructions, 14-1
instruction memory, 7-9
instruction prefetching, 7-10
instruction stream modification, 7-9 to 7-10
instruction-emulation traps
in a virtual machine, 12-15
INSV - insert field instruction, 3-46
integer
data types, 1-3, 1-6
divide-by-zero trap, 5-11
instructions, 3-7
overflow condition in vector processor, 13-10, 13-13, 13-33, 13-37 to 13-38
overflow trap, 5-11
overflow trap enable bit (IV), 1-27
vector instructions, 13-33
interlocking
changing page table entries, 4-12
effects on vector processor, 7-2, 14-36
in I/O space, 7-23
instructions, 7-6
restrictions on caches, 7-18
update of PTE modify, 4-11
internal processor registers (see IPR)
interrupt priority level (see IPL)
interrupt stack, 5-27
bit in the PSL (IS), 1-26
not valid, 5-19, 5-27
pointer (ISP), 5-27
interrupts, 5-2 to 5-3, 5-6 to 5-7, 14-30, 14-32 to 14-33, 14-36
AST delivery, 6-5
definition of, 5-1
initiation of, 5-31
precedence of, 5-21
process scheduling, 6-6 to 6-7
restrictions to allow restarting instructions, 5-5
software interrupts, 5-1, 5-7
vectors in the SCB, 5-6, 5-23 to 5-24, 5-28
interval clock, 8-22
registers, 8-23
interval clock control and status register (see ICCS)
interval count register (see ICR)
IOT - I/O trap instruction, 9-6, 9-54
IOTA - generate compressed iota vector, 13-58
IPL - interrupt priority level
as an IPR, 5-2, 5-9
definition of, 5-1
in the PSL, 1-26, 5-9
IPR - internal processor register
address space, 8-1
definition of, 1-24
implementation dependencies, B-3
implementation options, 11-4
in the PCB, 6-4
in the vector processor, 13-3, 13-11
IS - interrupt stack, 1-26
ISP - interrupt stack pointer, 5-27
IV - integer overflow trap enable, 1-27

JMP - jump instruction, 3-65, 9-43
JSB - jump to subroutine instruction, 3-66
JSR - jump to subroutine instruction, 9-44

kernel mode, 1-26
kernel stack, 5-27
  (see stack and stack pointer)
  not valid, 5-18, 5-27
  pointer (KSP), 5-27
KSP - kernel stack pointer, 5-27
  in the PCB, 6-4

LDPCTX - load process context instruction, 6-8, 14-41
leading-separate decimal string, 1-17
  (see also decimal string)
length violation, 4-16
  indication of, 4-32
literal mode addressing, 2-13
  restriction in vector processor, 13-16
LOAD console command, 10-23
LOCC - locate character instruction, 3-156
logical functions
  instructions, 3-7
  vector instructions, 13-39
longword
  data type, 1-6
  in registers, 1-22
  notation for, 3-3

M - PTE modify bit, 4-10 to 4-11
  affected by PROBEW, 4-35
machine checks, 14-36, 14-43
machine-check exception, 5-19
MAPEN - memory mapping enable register, 4-5, 4-28
MARK - mark stack instruction, 9-6
masked vector operations, 13-12
MATCHC - match characters instruction, 3-158
MAX testing, xi
MBZ, definition of, 1-2
MCOM - move complemented instructions, 3-23
memory
  use of, by vector processor (see memory, vector)
  address translation, 4-8, 4-16 to 4-17, 4-21 to 4-22, 4-26, 11-5, 14-41
  enabling memory mapping, 4-5, 4-28
  faults, 4-16, 4-30
  I/O address space, 7-18
  instruction memory, 7-9
  memory management, 4-1
  P0 and P1 regions, 4-21
  page, 4-2
  page boundaries, 4-16
  physical address, 4-6
  physical address space, 7-18, B-2
  process space, 4-2, 4-21 to 4-22, 4-26
  protection of, 4-7 to 4-8, 4-12 to 4-13, 4-16, 4-32
  required references, 7-14
  rtVAX process space, 11-6
  sharing of, 4-2, 7-1
  system space, 4-4, 4-16 to 4-17
  virtual address, 4-2, 4-4
  when mapping is disabled, 4-6 to 4-7, 14-42
memory management (see memory)
memory management exceptions, scalar (see exceptions)
memory management exceptions, vector, 14-15
  asynchronous MME handling, 14-16
  fault parameter, 4-32, 14-15
    PTE bit, 14-16
    VAL bit, 4-32, 14-16
    VAS bit, 4-32, 14-16
    VIO bit, 4-32, 14-16
  fault stack frame, 14-18
  synchronous MME handling, 14-16
memory mapping enable bit (MME), 4-5, 4-28
memory mapping enable register (MAPEN), 4-5, 4-28
memory, vector
  access mode, 13-23, 14-3
  accessing page tables, 14-42
  address translation, 14-41
  alignment, 13-23
  HALT considerations, 14-35
  instructions, 13-23
  intent to modify, 13-14
  required use of synchronization instructions, 14-34
  scalar/vector synchronization of, 14-30
  stride, 13-23
MFP - move from previous space instructions, 9-50
MFPR - move from processor register instruction, 8-5

  - vector IPRs, 13-3, 13-5, 14-20
  - VPSR, 13-3, 14-20, 14-34
- MFPS - move byte from PSW instruction, 9-6
- MFPT - move from processor instruction, 9-6
- MFVP - move from vector processor, 13-60, 14-2, 14-26
- MINU - minimum unsigned notation, 3-5
- MME - memory mapping enable bit, 4-5, 4-28
- MNEG - move negated instructions
  - floating point, 3-133
  - integer, 3-24
- mode
  - access, 1-26
  - CHM - change mode instructions, 5-40
  - compatibility, 1-26
  - native, 1-26
- modify access type, 3-2
  - (see also access type)
- modify bit of a PTE (M), 4-10 to 4-11
- modify fault, 5-13
  - caused by PROBEW, 4-35
- modify-fault, vector, 14-41
- MOV - move instructions, 9-26
  - floating point, 3-134
  - integer and logical, 3-25
  - move address, 3-37
  - move character string, 3-160, 3-164, 3-167
  - move complemented, 3-23
  - move IPR, 8-3, 8-5
  - move negated floating point, 3-133
  - move negated integer, 3-24
  - move zero-extended, 3-26
  - MOVPSL - move PSL, 3-88
  - packed decimal, 3-206
- MOVPSL - move PSL, 3-88
- MOVZ - move zero-extended instructions, 3-26
- MSYNC - memory instruction synchronization, 13-61, 14-26, 14-31, 14-34, 14-38
- MTP - move to previous space instructions, 9-49
- MTPR - move to processor register instruction, 8-3, 14-41
  - vector IPRs, 13-4, 14-41
- MTPS - move byte to PSW instruction, 9-6
- MTVP - move to vector processor, 13-62
- MUL - multiply instructions, 9-30
  - EMOD - extended multiply and integerize, 3-131
  - EMUL - extended multiply, 3-21
  - floating point, 3-135
  - floating point, vector, 13-53
  - integer, 3-27
  - integer, vector, 13-37
  - packed decimal, 3-208
- multiprocessors
  - instruction stream modification, 7-11
  - PTE - page table entry, 4-11 to 4-12
  - restrictions on caches, 7-18
  - sharing of data, 7-4
  - synchronization, 5-2, 7-1
  - visibility, 7-2

- N - negative condition code, 1-27
- native mode, 1-26
- NEG - negate instructions, 9-14
- negative condition code (N), 1-27
- NEXT console command, 10-24
- next interval count register (see NICR)
- NICR - next interval count register, 8-22 to 8-23, 11-4
- no-access
  - (see access type)
- NOP - no operation instruction, 3-89, 9-51
- numeric decimal string, 1-17
  - (see also decimal string)

- OA - operand address notation, 2-4
- octaword
  - data type, 1-6
  - in registers, 1-23
  - notation for, 3-3
- opcode
  - assignments, A-1
  - formats, 2-1
  - illegal vector, 13-17
  - vector assignments, A-8
- opcode reserved to customers fault, 5-14
- operand specifier, 2-3
  - notation, 3-2
  - restrictions for vector instructions, 13-16
- operand, atomic, 7-5
- overflow
  - condition code (V), 1-27
  - condition in vector processor, 13-10, 13-13, 13-33, 13-37 to 13-38, 13-43
  - fault, 5-12

trap, 5-11
overlapped vector instruction execution, 14-6
OWN - owner field of a PTE, 4-11

P0 and P1 registers
in the PCB, 6-5
P0BR - P0 base register, 4-22, 11-5
P0LR - P0 length register, 4-22, 11-5
P1BR - P1 base register, 4-26
P1LR - P1 length register, 4-26
restrictions when changing, 4-28
P0 or P1 space (see memory)
packed decimal string, 1-18
(see also decimal string)
page, 4-2
page boundaries, 4-16
page frame number field of a PTE (PFN), 4-10 to 4-11
page table entry (see PTE)
page tables, 4-1
paging of, 4-21, 4-32, 4-35
process page tables, 4-21 to 4-22, 4-26
restrictions when changing, 4-28, 6-5
rtVAX process page tables, 11-5 to 11-6
shadow, 12-18
system page table, 4-16
PC - program counter
definition of, 1-22
in the PCB, 6-4
PCB - process control block, 6-2
base register (PCBB), 6-1
PCBB - process control block base register, 6-2
PDP-11
(see also compatibility mode)
comparison of VAX with, 1-1
differences in interrupts, 5-2
PDP-11 (see compatibility mode)
per-process (see process)
performance monitor enable register (PME), 6-5
in the PCB, 6-5
restrictions when changing, 6-5
PFN - page frame number field of a PTE, 4-10 to 4-11
physical memory, B-1
(see also memory)
PME - performance monitor enable register, 6-5, 11-4
in the PCB, 6-5
restrictions when changing, 6-5
POLY - polynomial evaluation instructions, 3-137
POPR - pop registers instruction, 3-90
powerfail, 10-7, 14-35
autorestart, 10-3, 10-7
precedence
of memory management faults, 4-30, 14-16
of multiple events, 5-21
of trace fault, 5-15
previous mode (PRV_MOD), 1-26
priority level
(see IPL)
privileged-instruction fault, 5-13
PROBE - probe instructions
PROBER, PROBEW, 4-34
use of, 4-33
PROBEVMR, 12-9
PROBEVMW, 12-9
process
address space, 4-2
address translation, 4-21 to 4-22, 4-26
context, 6-1
context switching, scalar, 5-5 to 5-6, 14-2, 14-4, 14-35
context switching, vector, 14-3, 14-21
definition of, 6-1
page tables, 4-21 to 4-22, 4-26
paging of, 4-21, 4-32, 4-35
restrictions when changing, 4-28, 6-5
PCB - process control block, 6-2
regions, 4-2
rtVAX address translation, 11-6
rtVAX page tables, 11-6
rtVAX process space, 11-5
scheduling, 6-1, 6-6
processor access mode (see access mode)
processor identification register, 8-17
processor state, 1-22
processor status longword (see PSL)
processor status word (PSW), 1-23
processor type, 8-17
program counter (PC)
definition of, 1-22
in the PCB, 6-4
program I/O mode, 10-1
program region of process space, 4-2

PROT - protection field of a PTE, 4-10 to 4-11
protection
  protection of memory (see memory)
  PROT codes, 4-13
  PROT field of a PTE, 4-10 to 4-11
  protection codes in a virtual machine, 12-18
PRV_MOD - previous mode, 1-26
PRVCPU - previous cpu field, 6-5, 6-9
PSL - processor status longword
  access modes, 4-13
  changing the privileged half, 1-24
  correspondence between PSL and VMPSL, 12-5
  definition of, 1-23
  in the PCB, 6-5
PSW - processor status word, 1-23
PTE - page table entry, 4-8, 4-13
  for I/O devices, 7-19 to 7-20
  global page table index (GPTX), 7-20
  restrictions when changing, 4-12, 4-28
  when mapping is disabled, 4-7
PUSH - push instructions
  push address, 3-38
  PUSHL - push long, 3-28
  PUSHR - push registers, 3-91

quadruple precision floating point (see H_floating)
quadword
  data type, 1-6
  in registers, 1-23
  notation for, 3-3
queue instructions, 3-95

read-only access type, 3-2, 4-13
  (see also access type)
read-write access type, 4-13
  (see also access type)
real VAX, definition of, 12-1
register conflict, vector, 14-9
registers, 1-22
  (see also I/O registers)
  (see also IPR)
  base register, 1-22
  data types in, 1-22
  in compatibility mode, 9-53
  in the PCB, 6-4
  index register, 1-22
  interlocked operations on, 7-8
  values during bootstrap, 10-3
  vector control registers, 13-3
  vector internal processor registers, 13-3
  vector registers, 13-2
REI - return from exception or interrupt instruction, 5-37
relocation of memory (see memory)
REM - remainder notation, 3-5
REM - remove queue instructions
  REMQHI - remove from head interlocked, 3-105, 7-6 to 7-7, 7-15
  REMQTI - remove from tail interlocked, 3-108, 7-6 to 7-7
  REMQUE - remove, 3-111
REPEAT console command, 10-24
reserved
  definition of, 1-2
reserved addressing mode fault, 5-13
reserved instruction fault, 5-13
reserved operand
  condition in vector processor, 13-10, 13-43
  fault, 3-116, 5-13
  format, 3-116, 13-43
reserved to customers fault, 5-14
RESET - I/O reset instruction, 9-6
restart of system (see bootstrap)
restart parameter block (RPB), 10-5, 10-22
restartability of instructions, 5-3, 5-5 to 5-6
RET - return from procedure instruction, 3-77
ring compression, 12-16, 12-18
ROL - rotate left instructions, 9-22
ROR - rotate right instructions, 9-23
ROTL - rotate long instruction, 3-29
RPB - restart parameter block, 10-5, 10-22
RSB - return from subroutine instruction, 3-67
RTI - return from interrupt instruction, 9-47
RTS - return from subroutine instruction, 9-45
RTT - return from trap instruction, 9-47
rtVAX, 11-5
  address translation, 11-5

RXCS - console terminal receive
    control and status register,
    10-33, 11-4
RXDB - console terminal receive
    data buffer register, 10-33,
    11-4

SBC - subtract carry instructions,
    9-20
SBR - system base register, 4-16,
    4-28
SBWC - subtract with carry
    instruction, 3-30
SBZ, definition of, 1-2
scalar
  definition of, 1-2
scalar/vector memory
    synchronization, 14-30
SCANC - scan characters
    instruction, 3-169
SCB - system control block, 5-22,
    5-24
  implementation dependencies,
      B-2
  SCBB - system control-block
      base register, 5-23
SCC - set all condition codes
    instruction, 9-51
scheduling of a process, 6-1, 6-6
SEC - set C condition code
    instruction, 9-51
self test console command, 10-25
self-relative queue
  data type, 1-12
  definition of, 3-96
  instructions, 3-97
SEN - set N condition code
    instruction, 9-51
SET console command, 10-24
SEV - set V condition code
    instruction, 9-51
SEXT - sign extend notation, 2-4,
    3-5
SEZ - set Z condition code
    instruction, 9-51
shadow page tables, 12-18
sharing of memory (see memory)
shift instructions, vector, 13-41
SHOW console command, 10-25
SID - system identification
    register, 8-17
single precision floating point
    (see F_floating)
single step console commands,
    10-24
SIRR - software interrupt request
    register, 5-1, 5-9
SISR - software interrupt summary
    register, 5-7
SKPC - skip character instruction,
    3-171
SLR - system length register,
    4-28
SOB - subtract one and branch
    instruction, 9-41
SOBGEQ - subtract one and branch
    greater than or equal
    instruction, 3-68
SOBGTR - subtract one and branch
    greater than instruction,
    3-69
software interrupt, 5-1, 5-7
  request register (SIRR), 5-9
  summary register (SISR), 5-7
SP - stack pointer
  (see stack pointer)
SPANC - span characters
    instruction, 3-173
SPL - set priority level
    instruction, 9-6
SPT - system page table, 4-16,
    4-28
SSP - supervisor stack pointer,
    5-27
  in the PCB, 6-4
stack, 5-28
  (see also stack pointer)
  alignment, 5-28
  not valid, 5-27
  running on the interrupt stack,
      5-1, 5-27, 6-7
  switching between, 5-27, 5-31,
      5-38
stack frame
  CALL instructions, 3-71
  CHM instructions, 5-41
  emulation trap, 11-13
  machine-check exception, 5-19
  memory management fault, 4-30
stack pointer, 5-28
  (see also stack)
  definition of SP, 1-22
  in the PCB, 6-4
  internal processor registers
      (IPR), 5-28, 8-2
  switching between, 5-27, 5-31,
      5-38
state transitions of the system,
    5-36, 10-9
stride, vector, 13-23
string
  character string data type,
      1-17
  character string instructions,
      3-151

CRC instruction, 3-177
decimal string instructions, 3-183
EDITPC instruction, 3-216
in registers, 1-23
leading-separate string, 1-17
packed decimal string, 1-18
trailing numeric string, 1-17
SUB - subtract instructions, 9-28
floating point, 3-146
floating point, vector, 13-55
integer, 3-31
integer, vector, 13-38
packed decimal, 3-210
SBWC - subtract with carry, 3-30
subscript-range trap, 5-12
subsetting (see implementation options)
supervisor mode, 1-26
supervisor stack, 5-27
(see stack and stack pointer)
suspended emulation fault, 5-14, 11-11, 11-13
in a virtual machine, 12-16
SVPCTX - save process context instruction, 6-11
SWAB - swap bytes instruction, 9-24
SXT - sign extend word instruction, 9-21
SYNC - scalar/vector instruction synchronization, 13-60, 14-3, 14-29
synchronization, 7-1, 14-29
modifying the instruction stream, 7-9 to 7-10
of scalar/vector memory, 13-4, 13-61, 14-30
of vector exception reporting, 13-60, 14-29
restrictions on caches, 7-17
using IPL, 5-2
with I/O device registers, 7-23
synchronous memory management exception handling, 14-16
SYS_TYPE - System-type register, 8-17
system address space, 4-4
address translation, 4-16 to 4-17
system base register (SBR), 4-16, 4-28
system control block (SCB), 5-22, 5-24
base register (SCBB), 5-23
system identification register (SID), 8-17
system length register (SLR), 4-28
system page table, 4-16, 4-28
system state transitions (see transitions)
system type (SYS_TYPE), 8-17

T - trace enable, 1-27
TB - translation buffer (see translation buffer)
TEST console command, 10-25
time
in a virtual machine, 12-19
time-of-year clock register (TODR), 8-22, 11-4
TODR - time-of-year clock register, 8-22, 11-4
TP - trace pending, 1-26
tracing, 5-16, 5-18
breakpoint fault, 5-15
trace enable bit (T), 1-27
trace fault, 5-15
trace pending bit (TP), 1-26
trailing numeric decimal string, 1-17
(see also decimal string)
transitions
between major system states, 5-36, 10-1, 10-9
translation buffer (TB), 4-27
check register (TBCHK), 4-28
flushing before enabling memory management, 4-29
flushing entries for instruction stream modification, 7-11
invalidate (based on) ASN register (TBIASN), 4-28
invalidate all register (TBIA), 4-28, 14-41
invalidate single register (TBIS), 4-28, 14-41
vector invalidate all register (VTBIA), 13-4 to 13-5, 14-21, 14-24, 14-33, 14-41
within the vector processor, 13-5, 14-3, 14-41
translation of virtual addresses (see address translation)
translation-not-valid fault, 4-16, 5-13
trap
(see also exceptions)
definition of, 5-3
TRAP - trap instruction, 9-6, 9-54
TST - test instructions, 9-15
floating point, 3-148

integer and logical, 3-32
TXCS - console terminal transmit control and status register, 10-33, 11-4
TXDB - console terminal transmit data buffer register, 10-33, 11-4
type
  of processor, 8-17
  of system (SYS_TYPE), 8-17

UNDEFINED, 1-2
  definition of, B-3
underflow
  condition in vector processor, 13-10, 13-43
  fault, 3-117, 5-12
UNIBUS (see implementation options, UNIBUS)
UNJAM console command, 10-25
UNPREDICTABLE, 1-2
  definition of, B-3
user mode, 1-26
user stack, 5-27
  (see stack and stack pointer)
USP - user stack pointer, 5-27
  in the PCB, 6-4

V - overflow condition code, 1-27
V - valid bit of a PTE, 4-10 to 4-11
VADD - vector floating add, 13-45
VADDL - vector integer add, 13-33
VAER - vector arithmetic exception register, 13-4, 13-10
valid bit of a PTE (V), 4-10 to 4-11
validation of arguments, 4-33
variable length bit field
  (see bit field)
VAX architecture
  AXE testing, xi
  definition of, xi
  lists of implementation bugs, xiii
  MAX testing, xi
  testing conformance, xi
  who to contact, ii
VAX condition codes, 13-16
VAX implementations
  725, B-7
  730, B-7
  750, B-8
  780, B-8
  782, B-8
  785, B-8
  8200, B-9
  8300, B-9
  8500, B-9
  8600, B-9
  8650, B-9
  8700, B-10
  8800, B-10
  MicroVAX I, B-7
  MicroVAX II, B-7
VAXBI interconnect, 7-19
VBIC - vector bit clear, 13-39
VBIS - vector bit set, 13-39
VCMP - vector floating compare, 13-47
VCMPL - vector integer compare, 13-35
VCR - vector count register, 13-3, 13-60, 13-62
VDIV - vector floating divide, 13-51
vector
  definition of, 1-2
vector control registers, 13-3
vector control word, 13-12, 13-15, 13-18 to 13-21
  EXC - Exception Enable Bit, 13-13, 13-15 to 13-16, 13-18, 13-33, 13-37 to 13-38, 13-43, 13-46, 13-50, 13-52, 13-54, 13-56, 13-65, 14-15
  MI - Modify Intent bit, 13-14, 13-18, 13-25, 13-27, 13-65
  MOE - Masked Operations Enable bit, 13-12, 13-18, 13-65
  MTF - Match True/False bit, 13-12, 13-18, 13-65
  register specifier fields, 13-15
vector in the SCB, 5-23
vector instruction execution, 14-6
vector instruction formats, 13-12
vector processor disabled, 14-19, 14-21
vector registers, 13-2
VGATH - gather memory data into vector register, 13-14, 13-16, 13-26, 14-38
virtual address, 1-3, 4-2, 4-4
virtual machines
  PSL (see PSL and VMPSL)
  definition of, 12-1
  differences from real VAXes, 12-18
  emulation trap, 12-13
  initiation of exceptions from, 5-31
  modify fault required in, 4-9

option support of, 11-4
time in, 12-19
VVIA - VVAX interconnect adapter, B-10
VVII - VVAX I/O interconnect, B-10
VVMI - VVAX memory interconnect, B-10
virtual memory
(see memory)
size limits in a virtual machine, 12-19
virtual page number (VPN), 4-4
virtual VAX, definition of, 12-1
virtual-machine mode
bit in the PSL (VM), 1-26, 12-5
visibility of data, 7-2
VLD - load memory data into vector register, 13-14, 13-16, 13-24, 14-38
VLR - vector length register, 13-3, 13-60, 13-62
VM - virtual-machine mode, 1-26, 12-5
VM-emulation trap, 12-13
in CHMx, 5-41
in HALT, 3-85
in LDPCTX, 6-9
in MFPR, 8-5
in MTPR, 8-3
in PROBEx, 4-35
in REI, 5-38
VMAC - vector memory activity check, 13-4, 14-4, 14-32, 14-34, 14-38, 14-43
VMERGE - vector merge, 13-57
VMPSL, 12-5
created by REI, 5-38
VMR - vector mask register, 13-3, 13-60, 13-62, 14-9
VMUL - vector floating multiply, 13-53
VMULL - vector integer multiply, 13-37
VPN - virtual page number, 4-4
VPSR - vector processor status register, 13-3, 13-6 to 13-9
AEX - Arithmetic Exception Bit, 13-8, 14-4, 14-20 to 14-21, 14-23 to 14-24
BSY - Busy Bit, 13-3 to 13-4, 13-7 to 13-9, 14-5, 14-23, 14-31, 14-42 to 14-43
IMP - Impl. HW Error Bit, 13-8, 14-20 to 14-21, 14-23 to 14-24, 14-43 to 14-44
IVO - Illegal Vector Opcode Bit, 13-8, 13-17, 14-20 to 14-21, 14-23 to 14-24
MF - Memory Fault Bit, 13-7, 14-2, 14-16, 14-25
PMF - Pending Memory Fault Bit, 13-7, 14-2, 14-16, 14-23, 14-25
RLD - State Reload, 13-7, 13-9, 14-25
RST - State Reset Bit, 13-3 to 13-4, 13-7, 13-9, 14-23, 14-34
STS - State Store Bit, 13-7, 13-9, 14-23
VEN - Enable Bit, 13-3, 13-7, 13-9, 14-1, 14-4, 14-19 to 14-21, 14-23 to 14-25, 14-41, 14-44
VSAR - vector state address register, 13-4
VSCAT - scatter vector register data into memory, 13-14, 13-16, 13-30, 14-38
VSL - vector shift logical, 13-41
VSMERGE{F,D,G} pseudo-opcodes, 13-67
VST - store vector register data into memory, 13-14, 13-16, 13-28, 14-38
VSUB - vector floating subtract, 13-55
VSUBL - vector integer subtract, 13-38
VSYNC - synchronize vector memory access, 13-63, 14-34, 14-38
VTBIA - vector tb invalidate all, 13-4 to 13-5, 14-21, 14-24, 14-33, 14-41
VVAX, B-10
I/O space, B-11
Machine check exceptions
nonexistent memory, B-12
physical address space, B-11
read/write memory, B-11
VVCVT - vector convert, 13-49
VVIA - VVAX interconnect adapter, B-10
VVII - VVAX I/O interconnect, B-10
VVMI - VVAX memory interconnect, B-10
VXOR - vector exclusive-OR, 13-39

WAIT, 12-7
PDP-11 instruction, 9-6
use of in a virtual machine, 12-18

word data type
  definition of, 1-3
  in registers, 1-22
  notation for, 3-3
word-tearing, 7-4
write access type, 3-2, 4-13
  (see also access type)

X console command, 10-26
XFC - extended function call
    instruction, 3-92, 5-14
XMI interconnect, 7-19
XOR - exclusive-OR instruction,
    9-32
XOR - exclusive-OR instructions,
    3-33

Z - zero condition code, 1-27
zero condition code (Z), 1-27
ZEXT - zero extend notation, 2-4,
    3-5

digital™